[illegible]
NAG2-371
IN-25-CR
217300
[illegible]

Chemical calculations on Cray computers

Peter R. Taylor
ELORET Institute[†]
Sunnyvale, CA 94087

and

Charles W. Bauschlicher, Jr and David W. Schwenke
NASA Ames Research Center
Moffett Field, CA 94035

Note to printer:

Parts of computer programs are reproduced in this MS using a `TYPEWRITER`-like monospaced typeface. It is important that these be set using monospacing, and that the indentation and interword spacing in these blocks be reproduced as given. Also, we have made a careful distinction between the adjective "Cray" (always with first letter capitalized) and the computer designation "CRAY" (always all upper case). We have followed exactly the usage recommended by Cray Research Inc. and the text should be set following this usage.

(NASA-CR-183240) CHEMICAL CALCULATIONS ON CRAY COMPUTERS (Eloret Corp.) 110 p
CSCL 07D
N89-26061
Unclas
G3/25 0217300

[†] Mailing address: NASA Ames Research Center, Moffett Field, CA 94035

Abstract

The influence of recent developments in supercomputing on computational chemistry is discussed with particular reference to Cray computers and their pipelined vector/limited parallel architectures. After reviewing Cray hardware and software we examine the performance of different elementary program structures and outline effective methods for improving program performance. We then discuss the computational strategies appropriate for obtaining optimum performance in applications to quantum chemistry and dynamics. Finally, some discussion is given of new developments and future hardware and software improvements.

I. Introduction

The advent of supercomputers has had a profound influence on the development of computational chemistry in the last decade. Any increase in computing power from a given level will, of course, increase the range and size of problems that can be studied, but the influence of supercomputers goes much deeper than this. The need to develop new algorithms to exploit supercomputers fully affects formal mathematical aspects of computational chemistry methodology, while in some areas the ability to perform calculations at new levels of accuracy or on new chemical systems can alter entire computational chemistry strategies. In the present review, our goal is to show how the supercomputers produced by Cray Research Inc. (CRI) have influenced two areas of computational chemistry — molecular electronic structure and dynamics calculations. We shall discuss aspects of performance and programming for a range of Cray computers, and show how these factors have influenced the methodology and implementation in these areas. We shall also discuss how the results obtained from calculations have in turn influenced the philosophy behind the calculations.

At the time of writing, several different supercomputer models are produced by CRI. All are parallel, pipelined vector computers. The CRAY X–MP series is a development of the original CRAY–1: the machines are equipped with one to four CPUs, up to 16 megawords (MW) of very fast memory (up to 64 MW in the latest models) with multiple channels to each CPU, and a clock period of 8.5 ns (10 ns on the "se" series). The X–MP machines are characterized by excellent scalar performance and a rather rapid convergence to their asymptotic performance limit, that is, good performance on arithmetic involving short vectors. For example, they achieve over half their asymptotic performance of 235 million floating-point operations per second (MFLOPS) in multiplication of matrices of order 11×11. The CRAY Y–MP is a natural successor to the X–MP, with a clock period of 6 ns and up to 32 MW of memory. Its performance characteristics are very similar to those of the X–MP. The CRAY–2 is a rather different development, compared to the X–MP, from the original CRAY–1. The clock period is only 4.1 ns, and the machine can be equipped with up to 512 MW of relatively slow memory. As a consequence, at well over 400 MFLOPS the asymptotic performance is even higher than the Y–MP, but the performance on scalar or short vector arithmetic is not as good, proportionally, as on the X–MP series. This difference should not be overemphasized, however: the CRAY–2 still reaches half its asymptotic performance with the multiplication of matrices of order 25×25. In this sense all the Cray computers can be regarded as generally similar in achieving excellent performance without the need to go to long vector lengths. They are thus quite different from supercomputers like the CDC CYBER 205, or many of those from Japanese manufacturers.

Obtaining high performance from Cray computers requires the proper exploitation of parallelism in codes to use multiple functional units, pipelined vector hardware, and (where available) multiple CPUs. We can consider parallelism as arising at several levels. At the lowest level is the possibility of parallel execution of instructions derived from a single program statement, using the multiple functional units. At the next level is vectorization, the "parallel" execution of loop iterations using the vector functional units. From the programming point of view the fact this is not strictly parallel but pipelined execution is usually irrelevant. It is also possible to execute different loop iterations (or groups of iterations) on different CPUs. This next level of parallelism, termed "microtasking" by CRI [1,2], is more elaborate than the lower levels, as code must be included to acquire, release and possibly synchronize other CPUs. An even higher level of parallelism is the execution of larger code fragments (individual subroutines, say) on several CPUs simultaneously. This approach, referred to as "macrotasking" by CRI [1,2], may involve execution of quite different tasks on different CPUs, whereas microtasking would usually have all CPUs executing the same instructions but with different data. Finally, the highest level of parallelism corresponds to running separate user jobs on the various CPUs: this is conventional multiprocessing at the operating system level. We shall be particularly concerned with the implementation of parallelism at these various levels in computational chemistry codes in this review.

Our aim here is to explain how various methods of computational chemistry can be formulated to take maximum advantage of the power of Cray supercomputers. In order to provide the necessary background for this we discuss the characteristics and performance of different Cray computers, and then we consider a number of computational chemistry activities in some detail. Our emphasis will be on the utilization of Cray computer parallelism and we assume readers are already familiar with the formalism of *ab initio* electronic structure methodology [3] or of classical and quantum mechanical scattering [4-6]. We should also note here that as we routinely perform calculations using computers other than those from CRI, we generally avoid programming techniques that are very machine specific. In particular, we eschew the use of assembly language, and we generally attempt to implement algorithms that will be reasonably efficient on other supercomputer architectures. Some examples of algorithm choice motivated in part by these portability considerations are presented in our discussion.

In the next section, we discuss the various Cray computers, both hardware and software. In section III we describe the performance characteristics of the different machines for some typical code fragments, and deduce from these results the appropriate techniques for obtaining maximum performance. In sections IV and V we discuss the implementation of these techniques in various steps of molecular electronic structure calculations and dynamics calculations, respectively. These sections also review the ways in which supercomputers have influenced different aspects of computational chemistry. Section VI comprises our conclusions; we also speculate on the role forthcoming machines, such as the CRAY–3, will play in computational chemistry.

II. Cray computers

A. Hardware

We shall discuss the performance and use of the CRAY X–MP, CRAY Y–MP, and CRAY–2 computers. As stated above, these are pipelined vector processors with multiple high-speed CPUs. Vector arithmetic is performed on operands held in eight 64-element 64-bit vector registers. Results from one floating-point unit can be passed to other floating-point units while being returned to the registers on the X–MP and Y–MP: this is referred to as "chaining". This feature is not available on the CRAY–2. Our discussion of the X–MP is concerned primarily with a four CPU eight MW X–MP/48, an older machine with a clock period of 9.5 ns as opposed to the 8.5 ns of the latest X–MPs. This X–MP/48 features hardware GATHER/SCATTER and is configured with an eight MW input/output processor and a 128 MW solid-state storage device (SSD). The main memory is divided into 32 banks, and has a memory access time of eleven clock periods to load a single word into a CPU register. Successive banks can deliver words to the registers in successive clock periods, but each bank is busy for four clock periods after initiation of a read request. Each CPU has four channels to memory: two read, one write and one input/output (I/O). We shall also discuss some experience with an X–MP/14se , with a single CPU (clock period 10 ns) and four MW of memory. Comparisons between the performance of the X–MP/48 and the X–MP/14se are given below.

We shall discuss the performance of two slightly different CRAY–2 computers. Both are four CPU machines with 256 MW of main memory and a clock period of 4.1 ns. Each CPU has a single channel to memory, and 16 kilowords (KW) of very fast "local memory" that can be used as a type of cache to enhance performance. However, explicit control over local memory is available only to the assembly-language programmer, although high-level language compilers are being extended to generate code to utilize this hardware, as are some library routines. The CRAY–2 main memory is divided into 128 banks, but a software technique ("pseudo-double banking") provides emulation of 256 banks. Like the X–MP, successive banks can deliver words in successive clock periods. However, the CRAY–2 memory is, in effect, divided into quadrants, and in any particular clock period a given CPU can access only one of the quadrants. This causes significant complications in a multi-user environment, as when a user job recovers a CPU and starts to issue memory requests it may well find the requests "out of phase" with the CPU quadrant access. Even in dedicated mode it is possible for a job to be up to three clock periods out of phase with memory. The two CRAY–2s discussed differ in that the older machine has a bank busy time of 57 clock periods, while the newer machine (referred to below as CRAY–2*) has a somewhat reduced bank busy time of 42 clock periods. Clearly, there is a very significant difference between the memory performance on the CRAY–2 and on the X–MP and this difference appears in almost all performance comparisons, as we will see below. We shall also present results obtained on a CRAY Y–MP, equipped with 8 CPUs (6 ns clock period), 32 MW of memory (256 banks) and a 256 MW SSD. As expected, the Y–MP behaves very like the X–MP but scaled in performance by the faster cycle time.

Various disk subsystems are available from CRI; the machines to which we have access are largely configured with DD49 disk drives, with 150 MW of storage per drive and a data transfer rate on the order of 1 MW per second (MW/s) for a single unit. The X–MP and Y–MP have input/output (I/O) processors, essentially a large memory (8 MW on our X–MP/48, 32 MW on our X–MP/14se) buffer between CPU and disk. The transfer rate between the I/O processor and CPU is more than ten times faster than the transfer rate to disk, and substantial improvements in I/O performance can be achieved by "striping" files across multiple disks, so that successive blocks are written to different drives. These blocks can be read into the I/O processor in parallel, and then the data can be transferred to the CPU. Even better I/O performance can be obtained by using the SSD, available for the X–MP and Y–MP machines. The X–MP/48 we discuss has two channels to the SSD, and each is capable of transferring data at over 150 MW/s. In addition, the SSD has no overheads associated with head positioning and rotational delays, so that I/O can be performed with essentially no I/O wait time.

In the latest system versions I/O processing on the CRAY–2 also uses disk striping, and the larger memory of the CRAY–2 allows large buffers to be used for read-ahead/write-behind on sequential system I/O. Further, by taking advantage of this large memory when programming sorting steps it will often be possible to sort in memory, instead of using direct access disk I/O.

B. Software

1. Operating systems

There are three operating systems in common use for Cray computers: COS, UNICOS and CTSS. COS is a batch-job-oriented system that runs on the X–MP (and the CRAY–1). UNICOS is an interactive system for the CRAY–1, X–MP, Y–MP, and CRAY–2, based on AT&T UNIX System V with significant extensions (including some from Berkeley UNIX). CTSS is another interactive system, based on the Livermore Time-Sharing System. We will not discuss CTSS further here, nor will we consider the Guest Operating System, which allows UNICOS to be run on some CPUs of a multiprocessor system and COS on others. The X–MP/48 results we discuss are obtained under COS, while the X–MP/14se, Y–MP, and CRAY–2 results are all obtained under UNICOS.

COS, as noted above, is a batch-job-oriented system: it has a fairly limited repertoire of job control language (JCL) commands, but these nevertheless provide essentially all the functionality required to carry out large-scale scientific computations. In addition to compilers (discussed below) and loaders, COS [7] features program maintenance utilities, permanent file management and archiving, and dynamic (transparent to the user) management of resources such as the SSD or scratch file space for local files. However, CRI is apparently committed to UNICOS as its recommended operating system, and support for COS is being gradually withdrawn. For example, new sites are normally installed with UNICOS as the operating system. While this may have some advantages from the point of view of operating system maintenance (COS being written largely in Cray assembly language (CAL) and UNICOS largely in C), it is not clear how much benefit the end-user sees, especially the end-user interested mainly in large-scale scientific calculations. This issue is discussed at greater length below.

UNICOS [8] features essentially a complete set of UNIX commands, including both the Bourne and C shells and the TCP/IP remote file transfer (FTP) and communications (TELNET) package. UNIX itself is rather deficient in aspects of resource management and batch job execution (no queues, no scratch files, etc), and these deficiencies are being addressed as part of the development of UNICOS. For example, the Network Queueing System (NQS) [8] has been incorporated into UNICOS in order to provide control over job execution and queues. However, UNICOS version 4.0 (the latest release at the time of writing) does not yet provide as flexible a batch environment as COS, at least from the point of view of the production user, although future releases should improve this situation. It is also unfortunate that at the present time the X–MP and CRAY–2 run under slightly different dialects of UNIX, especially in view of the reputation of UNIX as a portable operating system. Presumably the differences will gradually be eliminated as UNICOS matures.

It may be useful here to compare UNICOS and COS as operating systems as they are seen by a user interested in large-scale scientific calculations. First, there is certainly some convenience in having interactive access to the machine from the point of view of compiling, debugging, etc, although the delay in turnaround for small jobs in going through, say, a VAX station to X–MP COS should seldom be significant. Further, as many of the UNIX commands were devised to assist in program development there is a wider range of powerful software tools available in UNICOS than in COS. On the other hand, from the point of view of running production jobs, UNICOS (strictly, UNICOS plus NQS) in its present version (4.0) appears to be inferior to COS. The dearth of resource management facilities in UNICOS (for example, the lack of any method for allocating and managing scratch file space beyond user goodwill in CRAY–2 UNICOS) makes the COS environment much more convenient for the production user. Further, comparisons suggest that user job throughput *on the same hardware* can be considerably (several times) higher using COS. This is related in part to the rather poor I/O performance we have experienced under X–MP UNICOS, where sequential I/O performance has been severely impacted when several jobs are running [9]. This affects CPU times as well as wall clock times, by a factor of four or more: we have observed a factor of ten increase relative to COS when I/O is done with the default file block size. Finally, at this time (using COS version 1.16) none of our production tasks reveal operating system bugs in COS that require workarounds or special tactics (the issue of compiler bugs is treated below), while even under UNICOS 4.0 certain codes will not operate correctly without special modifications. Again, this situation will no doubt improve as UNICOS matures, but at present our view is that computational chemists are better served by COS than by UNICOS.

2. FORTRAN compilers

As discussed in the Introduction, our aim is to avoid the use of assembly language and to use a high-level language for our codes. The debate over FORTRAN's deficiencies continues, but for most scientists the accumulation over a number of years of a considerable body of functioning software written in FORTRAN, together with the availability of optimizing compilers, makes it the high-level language of choice (see, e.g. Ref. 10).

The X–MP and the CRAY–2 each offer two compilers invoked as CFT and CFT77. It is a little unfortunate in view of the nomenclature that the CFT compilers on the two machines are not the same: CRAY X–MP CFT [11] is a product line that goes back to the CRAY–1, and has been fully compatible with the FORTRAN 77 standard since 1981 (version 1.10); the CRAY–2 compiler [12], whose product name is CFT2, is a FORTRAN 66-based compiler (again originally derived from an early version of the CRAY–1 compiler) that is still not compatible with the FORTRAN 77 standard. Both of these compilers are written in assembly language. CFT77 [13] is (in principle) the same product on the X–MP and the CRAY–2, and as its name suggests is fully compatible with the FORTRAN 77 standard. This compiler is written in Pascal, and as a consequence is much slower in compiling code than either version of CFT (sometimes by an order of magnitude or more), but this is only of consequence when large programs have to be completely recompiled. CRI's long-term commitment seems to be to CFT77: CFT is not available on the Y–MP, for example.

Both the CFT and CFT77 compilers feature extensive optimization and vectorization capabilities, as would be expected; CFT77 also includes explicit array syntax following the FORTRAN 8X proposal [10]. In addition, the newest release of CFT77 provides some capability for recognizing microtasking directives [1,2] and generating code to run on multiple CPUs. The compiler optimization involves both local and global (critical path) techniques similar in their effect to other advanced FORTRAN compilers. The compilers' ability to vectorize code has improved substantially over the years, especially in the area of conditional statements in loops, or generation of code for vectorized and non-vectorized versions of loops with run-time decision as to which is correct to use. While a number of constructs still inhibit vectorization, the compilers are being improved constantly in this area, and several examples are discussed later. As is common with vectorizing compilers, it may be necessary to inform the compiler explicitly that certain constructions should not inhibit vectorization, because from the code alone it may be impossible to tell whether vectorization should be allowed. In the loop

```
      DO 10 I = 1,N
         A(I+M) = A(I)
10    CONTINUE
```

the loop can be vectorized if $\mathtt{M} \geq \mathtt{N}$, but the compiler cannot check this at compile time, and would flag the loop as non-vectorizable. By including a directive informing the compiler that $\mathtt{M} \geq \mathtt{N}$ always holds, that is, to ignore any potential addressing problems — "vector dependencies" — the user can ensure the loop will be vectorized. In principle, a test for such dependencies could also be generated for execution at run time, but this facility is not currently available. The compiler directives [11-13] have the form of a FORTRAN comment statement, so this does not interfere with portability; however, while other manufacturers also employ such devices there is no standardization of directive names or syntax.

For efficient use of Cray computers, it is imperative to make maximum use of data once it has been loaded into the vector registers. The compilers are still not sophisticated enough to relieve the programmer of this job entirely: for example, the unrolling of DO-loops [14] usually enhances code performance under CFT or CFT77, unlike, say, the Alliant FX/FORTRAN compiler, for which user unrolling of loops generally *reduces* performance.

The latest improvements in CFT77 notwithstanding, FORTRAN support for using multiple CPUs on the various Cray machines is still fairly primitive [1,2]. Microtasking requires special constructs to be inserted in the code, which is then passed through a pre-processor to generate small CAL routines to manage the multitasking. It is this pre-processing step that the latest version of CFT77 appears to be able to manage for itself. Macrotasking, which normally involves coarse-grained parallelism, is handled through a set of FORTRAN-callable routines to organize starting multiple tasks and synchronizing them. In the very earliest form of microtasking, it was intended that microtasked versions of various library routines would be available, but this is not the case in the present release. This is a great pity, as a method of using multiple CPUs that was essentially transparent to the user (in particular, that required no non-standard code modifications) would be very valuable, as would automatic compiler-generated parallel code for DO loops (especially outer DO loops where inner DO loops have been vectorized). These facilities are already present in compilers and libraries for other machines, like the Alliant FX/8 [15,16], and will no doubt appear in the CFT77 compiler as it is further developed. Microtasked libraries are also promised by CRI. In view of the trend to larger numbers of CPUs (8 on the Y–MP, 16 on the CRAY–3) it seems imperative to provide strong support for multitasking — as the number of CPUs grows it will become increasingly difficult to keep a machine busy with individual jobs unless very sophisticated job schedulers are developed.

One of the inevitable consequences of exploiting machine parallelism is an increase in the memory required for a given computational task. One of the "8X" extensions to FORTRAN is the concept of allocatable arrays [10], storage for which is allocated dynamically on entry to a subroutine and which is deassigned (i.e. deallocated, at least in principle) on exit from the subroutine. This facility is available in recent versions of CFT77, but unfortunately the allocated storage is not returned on exit from the subroutine, so that the overall memory length can grow but not shrink using this approach. The use of allocatable arrays also incurs substantial overheads at present. More traditional methods of "dynamical allocation" are based on user-controlled expansion of the field length to adjust the size of blank common, which is conventionally loaded at the end of user memory. In this way memory can be acquired and returned as desired. However, this approach reflects a fairly primitive philosophy — that of using a user stack and assuming that no other part of the job will alter the field length. This assumption is vitiated at the outset in COS, for example, where system buffers are allocated at the highest user addresses, "growing" downwards as more files are opened. Care is therefore required to ensure that the user's expanding stack does not collide with the system. In general, with more sophisticated memory management features becoming available (like the heap-based allocation in UNICOS [17]) it seems preferable to avoid expanding blank common and to make more use of allocatable arrays. This is especially true in multitasked jobs, where special attention must be paid to avoiding conflicts in addressing data structures in different tasks. The use of allocatable arrays within each task is clearly a much simpler approach than trying to coordinate the expansion of blank common by more than one task. It is to be hoped that this very powerful feature will be fully implemented soon, as it will provide an important part of the support for multitasking.

In view of the sophistication required of a compiler that is to perform global optimization, vectorization, and some exploitation of multitasking, it is not entirely unexpected that the Cray compilers occasionally produce erroneous code. Much of this (especially under CFT2 on the CRAY–2) derives from attempts to optimize too large and complicated a code segment, and can be cured simply by decreasing the maximum size of a code block the compiler will try to optimize. Further, errors seldom cause a code to produce answers close to the correct values — they usually produce obviously incorrect results. Nonetheless, such errors are a considerable nuisance, as they can often be data-dependent, and will disappear in small, manageable test calculations, occurring only in large, expensive production runs. Unlike the situation with the UNICOS operating system, where there has been an overall improvement as new system versions have been released, more recent compilers have displayed more problems than older versions. This is particularly true of the X–MP CFT compiler. No doubt the adoption of CFT77 as the sole FORTRAN compiler will allow more effort to be concentrated on improved and error-free generation of code to exploit the various levels of parallelism.

modifications) would be very valuable, as would automatic compiler-generated parallel code for DO loops (especially outer DO loops where inner DO loops have been vectorized). These facilities are already present in compilers and libraries for other machines, like the Alliant FX/8 [15,16], and will no doubt appear in the CFT77 compiler as it is further developed. Microtasked libraries are also promised by CRI. In view of the trend to larger numbers of CPUs (8 on the Y–MP, 16 on the CRAY–3) it seems imperative to provide strong support for multitasking — as the number of CPUs grows it will become increasingly difficult to keep a machine busy with individual jobs unless very sophisticated job schedulers are developed.

One of the inevitable consequences of exploiting machine parallelism is an increase in the memory required for a given computational task. One of the "8X" extensions to FORTRAN is the concept of allocatable arrays [10], storage for which is allocated dynamically on entry to a subroutine and which is deassigned (i.e. deallocated, at least in principle) on exit from the subroutine. This facility is available in recent versions of CFT77, but unfortunately the allocated storage is not returned on exit from the subroutine, so that the overall memory length can grow but not shrink using this approach. The use of allocatable arrays also incurs substantial overheads at present. More traditional methods of "dynamical allocation" are based on user-controlled expansion of the field length to adjust the size of blank common, which is conventionally loaded at the end of user memory. In this way memory can be acquired and returned as desired. However, this approach reflects a fairly primitive philosophy — that of using a user stack and assuming that no other part of the job will alter the field length. This assumption is vitiated at the outset in COS, for example, where system buffers are allocated at the highest user addresses, "growing" downwards as more files are opened. Care is therefore required to ensure that the user's expanding stack does not collide with the system. In general, with more sophisticated memory management features becoming available (like the heap-based allocation in UNICOS [17]) it seems preferable to avoid expanding blank common and to make more use of allocatable arrays. This is especially true in multitasked jobs, where special attention must be paid to avoiding conflicts in addressing data structures in different tasks. The use of allocatable arrays within each task is clearly a much simpler approach than trying to coordinate the expansion of blank common by more than one task. It is to be hoped that this very powerful feature will be fully implemented soon, as it will provide an important part of the support for multitasking.

In view of the sophistication required of a compiler that is to perform global optimization, vectorization, and some exploitation of multitasking, it is not entirely unexpected that the Cray compilers occasionally produce erroneous code. Much of this (especially under CFT2 on the CRAY–2) derives from attempts to optimize too large and complicated a code segment, and can be cured simply by decreasing the maximum size of a code block the compiler will try to optimize. Further, errors seldom cause a code to produce answers close to the correct values — they usually produce obviously incorrect results. Nonetheless, such errors are a considerable nuisance, as they can often be data-dependent, and will disappear in small, manageable test calculations, occurring only in large, expensive production runs. Unlike the situation with the UNICOS operating system, where there has been an overall improvement as new system versions have been released, more recent compilers have displayed more problems than older versions. This is particularly true of the X–MP CFT compiler. No doubt the adoption of CFT77 as the sole FORTRAN compiler will allow more effort to be concentrated on improved and error-free generation of code to exploit the various levels of parallelism.

III. Performance

A. Elementary computational chemistry kernels.

A number of previous studies of Cray computer performance in the context of computational chemistry have appeared, and timing information on various types of constructs (primitive operations or "kernels", as they are termed in the study by Saunders and Guest [18]) that occur widely in computational chemistry codes have been given. In the present work we will use two typical kernels to illustrate particular performance aspects of different Cray computers. We will then briefly present performance figures of some other representative kernels. We should note here that the timing results we present are subject to some uncertainties, especially on the CRAY–2 where memory access delays from other user jobs can make observed times reproducible to no better than 10%.

The first kernel we will consider is the addition of a multiple of one vector to another, given in FORTRAN as

```
      DO 10 I = 1,N
         A(I) = A(I) + SCALAR*B(I)
10    CONTINUE
```

(referred to as SAXPY, using the BLAS name [19]). This is a rather typical simple vector operation and its behavior will serve as a model for other loops. The second kernel is the more elaborate operation of matrix multiplication, represented in FORTRAN (with a SAXPY inner loop) by the code

```
      DO 10 J = 1,N
         DO 20 I = 1,N
            C(I,J) = A(I,1)*B(1,J)
20       CONTINUE
         DO 30 K = 2,N
            DO 40 I = 1,N
               C(I,J) = C(I,J) + A(I,K)*B(K,J)
40          CONTINUE
30       CONTINUE
10    CONTINUE
```

Matrix multiplication is a particularly efficient operation on all Cray computers and it therefore offers the greatest scope for enhancing program performance. In practice, what is often desired is the more elaborate expression

$$\mathbf{C} = \alpha\mathbf{C} + \beta\mathbf{AB}, \tag{III.1}$$

which forms the proposed Level 3 BLAS specification [20]. We will discuss later the handling of the more general forms and of sparseness in the matrices.

For some simple kernels, it is possible to estimate the expected performance by considering instruction times. However, in most cases this is not very fruitful, as both compiled code and library code may take advantage of special features (or may be handicapped by special problems such as bank conflicts) and often produce quite different rates. We have chosen here to discuss only observed performance obtained on various machines (running a normal job mix, not in stand-alone mode) under different compilers. By analogy with the approach to hardware characterization of Hockney and Jesshope [21], we can evaluate from the performance data two quantities characteristic of each kernel: the maximum performance, r_∞, (a rate in MFLOPS) and the vector length at which half this performance is observed, denoted $n_{1/2}$. (We should note that since most operations do not yield monotonically increasing performance rates, r_∞ is not taken as the asymptotic rate, but rather as the highest observed rate for any vector length up to 1024.) In addition to the rather detailed discussions of SAXPY and matrix multiplication performance, we tabulate r_∞ and $n_{1/2}$ values for a number of other kernels concerned with both arithmetic and data motion.

B. SAXPY and matrix multiplication.

Table 1 shows the performance for the SAXPY operation obtained on a CRAY X–MP/48, using the CFT and CFT77 compilers, and also the SAXPY subroutine from CRI's scientific subroutine library, SCILIB [17,22]. It is evident that the best performance is obtained using the CFT77 compiler, avoiding the overhead associated with calling the library routine. CFT77 also displays the smallest $n_{1/2}$ value. The better performance of the code obtained from CFT77 relative to CFT is rather typical of the behavior of the two compilers (at least at the time of writing): CFT77 commonly produces code faster by 20 to 30% or more. All times in Table 1 show a steady growth in performance as the vector length increases, up to a vector length of 63. Somewhat unexpectedly, in view of the "natural" vector length of the machine, the performance for vector lengths of 64 is not quite as high as for 63, although this may reflect the fact the code generated is designed to cope with vectors longer than 64. This behavior is not observed for other pairs of lengths such as 127 and 128, or 255 and 256. Finally, we note that the best rate observed for the SAXPY operation falls short of the ultimate performance (210 MFLOPS for an X–MP with 9.5 ns clock), even though both an addition and multiplication are required in the innermost loop and these operations can be chained on the X–MP.

A comparison of SAXPY performance on several different Cray computers is given in Table 2. Only the approach that gives the highest r_∞ is shown: this is the simple FORTRAN loop of the previous subsection compiled with CFT77 on the X–MP machines and the SCILIB version of the BLAS call on the CRAY–2 machines. This difference probably reflects the absence of hardware chaining on the CRAY–2, which could be partially compensated for by careful assembly language coding of a library routine but which is probably beyond the compiler's capabilities. The best performance on the CRAY–2 is much less than the theoretical 488 MFLOPS, so even less of the machine's power can be exploited this way than on the X–MP. Also, while the CRAY–2 performance improves steadily up to vector lengths of about 64 there are numerous fluctuations above this value that do not correlate with any obvious hardware characteristics. The timing tests were performed on production machines in multi-user mode, so these fluctuations may represent problems with bank conflicts engendered by other user jobs and by context switching.

The X–MP/48 matrix multiplication results displayed in Table 3 provide a different perspective on performance. Here, the SCILIB routine MXM outperforms all the FORTRAN implementations listed. The latter comprise the SAXPY inner loop form given explicitly in the previous subsection, a similar approach with a dot product inner loop, and the SAXPY inner loop form as above with the J loop unrolled four times. Although the unrolled loop structure gives quite good performance with larger array dimensions, it is still not competitive with the library MXM. The latter not only has an r_∞value of almost 200 MFLOPS, but obtains half this rate with a matrix dimension of only 11. The use of the library matrix multiplication routine would thus seem to be an ideal route to high performance on the CRAY X–MP, even for problems of rather small dimension.

A comparison of matrix multiplication performance on several Cray computers is given in Table 4. For all machines the highest rates are obtained with the library MXM routine. While the CRAY–2 performance still falls somewhat short of the maximum possible, experience has shown that this derives substantially from bank conflict problems created by competition with other user jobs. In stand-alone mode we have observed CRAY–2 MXM performance of more than 420 MFLOPS. This is similar to the fraction of the theoretical performance obtained on the X–MP/48. Although $n_{1/2}$ for MXM on the CRAY–2 is larger than for the X–MP machines the values are still fairly small, so we can conclude that matrix multiplication is a route to high performance on the CRAY–2 as well as on the X–MP.

It is quite straightforward to rationalize the different behavior of SAXPY and matrix multiplication. In the latter, considerably more arithmetic operations can be done on data once it has been read from memory into the vector registers as compared to the former. There is therefore a much higher proportion of floating-point operations in the overall operation count in the matrix multiplication case and consequently higher performance. This observation is the key to programming Cray computers for high performance: to seek tasks which re-use data in the vector registers repeatedly, rather than using it once or twice. Matrix multiplication is probably the most obvious example of this approach, and we will generally try to show how it can be incorporated into user programs. Other related kernels, such as solving systems of linear equations, are also considered when we discuss particular computational chemistry tasks. In view of this central importance of matrix multiplication, we shall explore some aspects of it further here.

Reference has already been made to more general matrix multiplication operations such as (III.1). Although this form is not available in the SCILIB library, not even in the simple form involving the constraints $\alpha = 1$ (or 0) and $\beta = \pm 1$, several groups have prepared subroutines to perform this task. For the CRAY–2 the subroutine MXMPMA by Calahan *et al.* [23] handles the above form with the constraints $\alpha = 1$ or 0 and $\beta = \pm 1$. This is a very efficient implementation that exploits local memory on the CRAY–2. Another aspect of more general matrix multiplication is the exploitation of sparseness in the arrays **A** and **B**. Where sparseness has its origins in the symmetry of a problem, it is usually more advantageous to handle the symmetry explicitly, but it may often be useful to take advantage of essentially random sparseness. Saunders' routine MXMB [18] is widely used in this situation: its performance is similar to the CRI MXM routine for dense matrices so there is no loss of efficiency where there is little sparseness.

Another generalization of the simple matrix product is the case in which either **A** or **B** is to be transposed before the multiplication. The SCILIB routine MXMA allows for this possibility (and for other variations on the indexing of the matrices, such as multiplication of sub-blocks of full arrays). The alternative approach would be to explicitly transpose the arrays as required. On the CRAY X–MP/48 it is our observation that there is little to choose between these strategies. On the CRAY–2 for most array dimensions there is also little difference, but there is a catastrophic loss of performance in MXMA when transposing arrays with dimensions that are multiples of 256 because of severe bank conflicts. In such cases the observed performance drops to some 18 MFLOPS, about 20 times slower than the best rate. Several smaller dimensions, such as 128 or 64 also show a drop in performance, although not as great. While this can be overcome by embedding the desired matrix in a larger array to avoid the critical stride value, we normally prefer to explicitly transpose the matrix and use the routine MXM. Since, as noted, this strategy incurs virtually no penalty on the X–MP, we follow it on all Cray computers. This also increases the portability of programs, as it usually easier to find an analog of MXM on other machines than MXMA.

The technique of obtaining increased performance on Cray computers by unrolling an outer loop is well documented [14], but it is perhaps less obvious that this approach can be extended to more than one loop. In the matrix multiplication case this gives the following code fragment when two loops are unrolled to a depth of two:

```
      DO 10 J = 1,N,2
         DO 20 K = 1,N,2
            DO 40 I = 1,N
               C(I,J) = C(I,J) + A(I,K)*B(K,J) + A(I,K+1)*B(K+1,J)
               C(I,J+1) = C(I,J+1) + A(I,K)*B(K,J+1) + A(I,K+1)*B(K+1,J+1)
30          CONTINUE
20       CONTINUE
10    CONTINUE
```

Unrolling the J loop enhances performance by reducing the number of fetch instructions required for the elements of **A**, while unrolling the K loop reduces the number of fetch and store instructions for the elements of **C**. As **A** and **C** are different arrays, these advantages are combined when both loops are unrolled, and the overall efficiency is increased. Formally, the efficiency increases as the unrolling depth increases, but the finite number of vector registers, together with the limits on the size of code block the compilers can optimize imposes an upper limit to the useful depth of eight. CRAY–2 timings obtained with such a scheme (both loops unrolled eight times) are given in Table 5. Although the performance is considerably improved over the case of FORTRAN loops without unrolling, it is never as good as with MXM, so the latter would usually be preferred. An exception might be the case of equation (III.1) with α different from zero or one. The unrolled scheme can handle this case without the need to store a temporary product matrix separately, as would be required if MXM was used.

For very large matrices there is some advantage in using specialized matrix multiplication algorithms that require fewer than $2n^3$ floating-point operations [24-26]. Many such algorithms use a recursive partitioning approach akin to Fast Fourier Transforms, in which the matrices are multipled by blocks using expression rearrangement to eliminate some operations. The best of these schemes behaves as roughly $n^{2.5}$ [26], but a simpler scheme, due to Strassen [24], which behaves as about $n^{2.8}$, has recently been investigated on the CRAY–2 by Bailey [27]. We can compare this approach with the library routines and also with loop unrolling in FORTRAN. The results are displayed in Table 5. The routine MXMPMA is the assembly language program written by Calahan *et al.* referred to above [23]. It is this routine that is used to perform the block matrix multiplications in the Strassen scheme. The performance figures for the latter are computed assuming $2n^3$ floating-point operations: for the large matrices (say, larger than 500×500) they are distinctly better than the other values. Note also that MXMPMA does not suffer from the bank conflicts that somewhat degrade the library MXM performance for the 256×256 and 512×512 cases.

C. General performance.

Observed performance, given as r_∞ and $n_{1/2}$ values, for a number of operations are given in Table 6. The SAXPY and matrix multiplication results of the previous subsection are also reproduced for reference. For elementary vector operations like addition, the code produced by the CFT77 compiler out-performs the library subroutines on the X–MP and Y–MP, but the library versions run faster on the CRAY–2. Performance figures are also given for sparse vector operations "SPDOT" and "SPAXPY" [22]. These allow for sparseness to be exploited in one vector: the SPAXPY for example is equivalent to the FORTRAN loop

```
      DO 10 I = 1,N
         A(INDEX(I)) = A(INDEX(I)) + SCALAR*B(I)
10    CONTINUE
```

and is useful when vector B has been compressed down to non-zero values whose original addresses are stored in INDEX. SPDOT is simply the dot product equivalent. These kernels are useful in quantum chemical calculations: the performance of the library versions is quite good.

Polynomial evaluation using Horner's rule, such as

```
      DO 10 I = 1,N
         A(I) = ((C3*X(I) + C2)*X(I) + C1)*X(I) + C0
10    CONTINUE
```

for the cubic case, is very efficiently vectorized by CFT77 on the X–MP and Y–MP, where for higher polynomial orders a substantial fraction (about 90%) of the maximum machine performance is obtained. This indicates that the compiler is able to generate code to re-use data from the vector registers for this case. Polynomial evaluation is a less fruitful approach on the CRAY–2, where the convergence with increasing polynomial order is much slower.

Element-by-element vector division is also shown in Table 6. Cray computers have a reciprocal approximation unit but no floating-point divide hardware, so division requires a reciprocal approximation, a multiplication to obtain an estimate of the quotient, followed by a Newton iteration to give the required accuracy in the quotient (see, e.g. Ref. 21). This step requires a subtraction and two more multiplications. Even if the subtraction is chained with one multiplication this process cannot run at more than 40 MFLOPS (strictly speaking at 40 megaresults per second) on our X–MP/48, and the observed performance is noticeably less. However, if the division is part of a more elaborate computation, for which data can be held in the vector registers and re-used, the overall performance will be more satisfactory. The [2/1] rational fraction evaluation requires the same number of simple additions and multiplications as the cubic polynomial, and also requires division, with its contributing multiplications and subtractions, but the observed performance is still about two-thirds of the polynomial rate on the X–MP and Y–MP, and is equal to the polynomial rate on the CRAY–2.

For many purposes it is necessary to evaluate special functions, and the performance of the FORTRAN library routines for several of these is shown in Table 6. The rates here are quoted in megaresults per second, and as these functions can require 20 or more floating-point operations for their evaluation the performance is seen to be rather good. It is interesting that this is one of the few areas in which the CRAY–2 outperforms the X–MP. On all machines the square root performance is substantially better than the other functions but is still not competitive with elementary vector operations.

The last type of operation considered in Table 6 is data motion. On the CRAY–1 data motion was an expensive task, with a performance for a simple vector move of about 8 MW/s. With improvements such as hardware GATHER/SCATTER all of the current Cray computers perform much better than this, with vector move rates that are higher than operations like addition. Data motion on the CRAY–2 shows a somewhat worse performance ratio to the X–MP than that seen for elementary vector operations.

D. Multitasking on Cray computers.

As discussed in the Introduction, vectorization is not the only way to improve performance on those Cray computers with several CPUs: it is also possible to use several CPUs in parallel. The pros and cons of this approach, especially in the context of multi-user systems, have been discussed elsewhere [28]. Multitasking involves some overheads, so the *total* CPU time of a job will *increase* when it is multitasked [1,2]. Any timing improvement therefore comes (in a stand-alone machine) from a reduction in wall clock time. Except for the case of large user jobs that would be feasible only if run multitasked, where there is obviously no alternative, system throughput on a production machine is generally best served by minimizing user multitasking and maximizing multiprocessing (that is, "system multitasking"). There can be circumstances that demand a different approach, however. For example, we have previously discussed the "$1/n$ rule" [28]: on a system with n CPUs there will be no risk of CPUs becoming idle because of lack of system resources provided users are allowed to acquire no more than $1/n$ of any resource. Clearly, if a user job *requires*, say, all of main memory the user should be encouraged to use all the CPUs, in order to keep the entire machine busy. This is thus another motivation for investigating multitasking.

We concentrate here only on the performance obtained with multitasked codes. First, it should be noted that in any multiprogrammed system the opportunity to use multiple CPUs will vary considerably with workload and the various tuning parameters in the job scheduler. As a consequence, the performance measured for a multitasked job will commonly vary by a considerable amount from run to run, making it difficult to compare different approaches and different machines. Some of the results presented here were therefore obtained during dedicated access to a given machine. Second, multitasking is not yet a commonplace activity on Cray computers, at least in our observation, and therefore some aspects of multitasking have yet to be properly debugged. Finally, we should point out that, as a corollary of using dedicated time, some performance figures for the CRAY–2 are quite different from those given in the previous section. The reasons are discussed below.

Table 7 contains performance figures for matrix multiplication using different numbers of CPUs on different machines. This is a macrotasked matrix multiplication in which the result matrix is divided into n column groups (for n tasks), as described in detail in Ref. 28. The X–MP and Y–MP display an essentially linear scaling with the number of CPUs employed, producing the impressive figure of more than 2.3 GFLOPS when eight CPUs are used on the Y–MP. The overheads associated with generating the tasks and acquiring CPUs are not included in Table 7, but if the matrix multiplication is repeated several times within the job, as would probably be the case in a production code, the mean overhead becomes negligible anyway. The CRAY–2 figures scale considerably worse than n, even in a stand-alone environment. This has been discussed at length elsewhere: it stems from bank conflicts in one task generated by the other tasks. In a production environment the performance is even worse, as more bank conflicts arise from other user jobs and from the context switching and job swapping that occurs in multi-user mode. It is this interference between tasks that makes such a difference between stand-alone and multi-user mode times for even single-tasked jobs on the CRAY–2; it is noteworthy that our stand-alone multitasked matrix multiplication performance results on four CPUs are about four times the multi-user mode single-tasked results of section IIIB.

There is little value in repeating the above figures for microtasked versions of the matrix multiplication. These show essentially the same performance improvements as does macrotasking in a stand-alone environment, while the improvement possible in a production machine is an even stronger function of the workload than is the case for macrotasked jobs, so there is no simple way to quantify the performance observed. Microtasking is designed to utilize idle CPUs, and under UNICOS this appears to be implemented by giving extra tasks forked a very low priority (a high "nice" value, in UNIX terminology [29]). This means that these forked tasks will get a very small share (although not a zero share) of the CPUs, unless the number of user jobs in the system falls to such a level that only the forking task and its children remain, in which case all will utilize the CPUs. Obviously, in a dedicated environment a single microtasked job will utilize all CPUs fully and perform like its macrotasked equivalent, while in a normal production environment the forked tasks will accumulate almost no time and the result will be equivalent to the single-threaded version. On the other hand, an "idle CPU" under COS seems to correspond to a CPU not busy with a task with the same priority as the spawning task. Hence a high-priority job under COS will generally be able to acquire CPUs as it desires, greatly improving its throughput (and degrading that of other user jobs), while a similar job run at lower priority will see much less improvement (if any). COS microtasking can thus impact other users more than UNICOS microtasking, and for many COS production environments microtasking may be discouraged.

Overall, while multitasking can often improve performance, its use can be counter-productive in a multi-user environment. Under COS, at least, it appears desirable to have about eight jobs per CPU in the machine to ensure that no idle time accumulates, and at present (with Cray computers of up to four CPUs capable of running COS) it should be possible in most production environments to keep a machine busy simply by multiprocessing user jobs. This might not be the case with 16 or even more CPUs, as the sophistication required in a job scheduler to cope with keeping so many CPUs busy would be considerable. It may also be the case that for some environments in which very long jobs or jobs with very heavy resource requirements are run that it is not possible to schedule jobs to keep all CPUs busy. Such situations are probably better suited to the use of multitasking, at least for some part of the time. A useful compromise for many users would be to have microtasked versions of various library routines (especially matrix multiplication) available, at least under UNICOS where there is less impact of microtasking on other users. In this way the multitasking would be transparent to the user, but any idle CPUs could be utilized effectively. As noted above microtasked libraries are promised by CRI.

IV. *Ab initio* quantum chemistry

A. General observations

Many aspects of *ab initio* quantum chemistry program implementations for Cray computers have been discussed elsewhere [18,30]. We concentrate here on illustrating the general principles outlined above for efficient use of Cray computers, with examples from the MOLECULE [31,32] and SWEDEN [33] program systems. We discuss the evaluation and sorting of integrals, the optimization of SCF, MCSCF, and CI wave functions, and also several less conventional methods for different types of electronic structure calculations. As several authors have emphasized [34-36], the calculation of energy derivatives (gradients, Hessians etc) shares many features with the calculation of the energy itself, and for exigencies of space we shall only discuss the latter here. Most of the techniques described can readily be implemented in derivative calculations as well.

There is no intention here to instruct the reader in the details of efficient programming; the "Optimization Guide" published by CRI for the X–MP series [37] can be consulted for such material. We shall discuss some of the broader aspects of programming Cray computers, again under the assumption that the codes to be written must be fairly easily ported to other environments.

B. Gaussian integrals and integral sorting.

Several reviews of methods for evaluating Gaussian integrals, with special emphasis on vectorization, are available [38-40]: the comprehensive and lucid article by Saunders [38] is particularly recommended. The evaluation of one-electron integrals is straightforward and requires little time, and we deal only with two-electron integrals here. We shall concentrate here on special features of the MOLECULE integral program that are not covered by these reviews. MOLECULE evaluates integrals over symmetry-adapted linear combinations of contracted Gaussian functions; these contractions can be of either general [41] or segmented [42] type. Basis functions can be either Cartesian Gaussians or spherical harmonic functions — in the latter case it is possible to include not only "pure" spherical harmonics ($1s$, $2p$, $3d$, $4f$) but also the higher principal quantum number "contaminant" functions $3s$, $4p$, etc. As the previous reviews of integral generation have only sketchily treated general contractions, symmetry adaptation and transformation of Cartesians to a spherical harmonic basis (note that Saunders discusses in detail the calculation of integrals directly in a spherical harmonic basis), we shall treat these matters in some detail here. General contraction and transformation to spherical harmonics are both rather time-consuming, but the major part of the overall time is usually spent evaluating primitive integrals.

The algorithm actually used for evaluation of primitive two-electron integrals in MOLECULE is based on the traditional factorization [43] into a product of terms each involving one Cartesian direction, and an "auxiliary function" term that must be evaluated by numerical approximation, symbolically

$$I = \sum_{m} X_m Y_m Z_m F_m(t) \tag{IV.1}$$

where m runs from zero to a limit given by the sum of angular quantum numbers involved; $F_m(t)$ is the "auxiliary function", t depends on the relative distance between the various basis functions and their exponents. The efficient implementation of this approach requires that all functions from a given shell (that is, that differ only in azimuthal quantum numbers) be treated together. The factors X_m can be obtained by recursion, as shown originally by Boys [44] and as exploited in various forms by McMurchie and Davidson [45] and, recently, by Obara and Saika [46]. The recursion scheme used in MOLECULE has many features in common with the latter approach, although it was originally derived some years ago to help implement efficiently the integral formulas given by Huzinaga and coworkers [43]. Like most integral algorithms, this lends itself to "extrinsic vectorization" [18], that is, a suitable number of quadruplets of function exponents (in effect, a list of t values) are treated together, so the various manipulations involved in (IV.1) are performed for vectors of the desired length. This approach to vectorization is simplest if the terms grouped together are on the same set of four centers, so that when there are only one or two functions of a given type on each of the centers the vector lengths would be very short. The use of large primitive sets is becoming more common, however, and this produces satisfactory vector lengths in most cases. The values in the vector of t values vary over a wide range in most calculations, and there is considerable sparseness to be exploited in the use of (IV.1). It is convenient to compress the vector to only non-zero values, compute the auxiliary function values for the compressed list and then expand the result list back for use in (IV.1). This can be done in practice without requiring an index vector of the same length as the vector of t values because the latest version of X–MP CFT can vectorize the loop:

```
      J = 0
      DO 10 I = 1,N
         IF (A(I) .GT. THR) THEN
            J = J + 1
            B(J) = A(I)
         ENDIF
10    CONTINUE
```

Since the vector of t values can number upwards of 50 000 terms for a large basis set, the elimination of the index vector can be very valuable on the X–MP. The EXPAND operation that is the reverse of this compression is also vectorized by X–MP CFT.

Given then a set of primitive integrals, how can these be contracted efficiently using general contractions? This is, in effect, a four-index transformation

$$(ij|kl) = \sum_p \sum_q \sum_r \sum_s (pq|rs) T_{pi} T_{qj} T_{rk} T_{sl}, \qquad \text{(IV.2)}$$

but in cases where, say, six to ten primitives are contracted to two or three contracted functions, the vector lengths in (IV.2) are simply too short for effective performance when formulated in the conventional matrix multiplication scheme (discussed in section IVD below). It is preferable to use a different approach for each of the four partial sums in (IV.2): if the $(pq|rs)$ are arranged in a rectangular matrix **I** with column index p and a "compound" row index qrs, the first partial sum for (IV.2) can be written

$$I'_{qrs,i} = \sum_p I_{qrs,p} T_{pi}. \qquad \text{(IV.3)}$$

For N primitive functions and n contracted functions this is, in effect, a matrix multiplication of an $N^3 \times N$ matrix by an $N \times n$ matrix [47], where the more conventional scheme would have rs fixed and would thus involve only $N \times N$ matrices. As the Cray library matrix multiplication routines are organized to perform the work in the order that gives maximum vector lengths, the scheme (IV.3) will be vectorized with an inner loop of length N^3. The next partial sum requires a "transposition" of the result array $\mathbf{I}'$ to give indices q and the compound rsi, but this can be done straightforwardly either by an explicit transposition loop or, implicitly, by using the Cray library routine MXMA (see section IIIB above) to multiply matrices with non-unit stride between elements. For the CRAY-2, as discussed above, the explicit transposition has the virtue of reducing the performance losses arising from bank conflicts: no extra memory is required for the transposition as the transposed $\mathbf{I}'$ can overwrite the original $\mathbf{I}$. The simplest solution is thus to use MXM throughout.

Once the contraction has been performed for all sets of angular quantum numbers in a batch, it is possible (if desired) to transform to a basis of spherical harmonic functions. There are several advantages to such a transformation. First, and perhaps most importantly, if only the pure spherical harmonics are used the dimension of the basis (and therefore the length of the integral file) is reduced. Even the decrease in d shells from six components to five can bring about a useful reduction in the length of an integral file, while for a diatomic molecule the elimination of the contaminants from a $[5s\ 4p\ 3d\ 2f\ 1g]$ basis will reduce the length of the integral file by considerably more than half. Again, the size of the matrices that are involved in this process is rather small, especially for the commonest higher angular momentum cases (d and f shells), so that it is advantageous to use the same scheme as is used for the contraction transformation. However, the arrays of transformation coefficients from Cartesians to spherical harmonics are rather sparse, especially for the lower angular quantum numbers, so it is useful here to have a matrix multiplication scheme that can exploit sparseness. Nevertheless, even with the Cray library matrix multiplication routines the work associated with the transformation to spherical harmonics is a rather small fraction of the total integral time [32]. Incidentally, it may be useful to retain the contaminants (and perhaps even eliminate the high angular spherical harmonics) if desired, as methods that need continuum-like functions may require fewer functions of $8p$-type to describe a p-wave than they would of the usual $2p$ functions.

While the transformation from contracted Gaussian functions to symmetry-adapted linear combinations (restricted in MOLECULE and in our discussion here to D_{2h} and its subgroups) could be regarded as a four-index transformation, various formal simplifications make an alternative approach preferable. Most symmetry-adaptation procedures [48,49] can be viewed as implementations of the method of double coset decompositions presented by Davidson [50].

A symmetry-adapted integral is given as

$$(p_\alpha q_\beta | r_\gamma s_\delta) = N \sum_i \sum_j \sum_k \sum_l \chi^\alpha(g_i)\chi^\beta(g_j)\chi^\gamma(g_k)\chi^\delta(g_l)(\hat{g}_i p\ \hat{g}_j q | \hat{g}_k r\ \hat{g}_l s), \quad \text{(IV.4)}$$

where α... indexes irreducible representations, $\hat{g}_i$... are operators from the point group $\mathcal{G}$, p... are atomic basis functions, and N is a constant involving both normalization of the symmetry orbitals and selection rules on symmetry species. This expression can be replaced [50] by the simpler form

$$(p_\alpha q_\beta | r_\gamma s_\delta) = N' \sum_J \sum_K \sum_L \chi^\beta(g_J)\chi^\gamma(g_K)\chi^\delta(g_L)(p\ \hat{g}_J q | \hat{g}_K r\ \hat{g}_L s). \qquad \text{(IV.5)}$$

Here N' now incorporates the rules which select the unique α given the three other irreducible representations. It should be noted first that the number of summations in (IV.5) is three, compared to four in (IV.4): (IV.5) clearly involves less work. Second, the range of J, K and L in (IV.5) is easily restricted so that only unique atomic integrals need be computed and employed in obtaining the symmetry-adapted integrals [48,50]. (IV.5) can be vectorized very simply: as a group of atomic integrals with the same angular properties and centers are multiplied with the same factors in (IV.5), each term in the triple summation can be viewed as an element in a SAXPY operation. Thus the "symmetry transformation" can be vectorized as a set of SAXPY operations with length n^4 (in the above notation). In fact, by forming the symmetry integrals for multiple quadruplets of irreducible representations at the same time, the atomic integral values can be re-used several times once they have been read into registers, further improving the efficiency.

It can be seen from the foregoing paragraphs that the various operations we have discussed will involve either considerable data motion, or the use of non-unit strides. Thus the contraction of primitive integrals requires the processing of quadruplets of primitives for each quadruplet of angular quantum numbers, while the transformation to spherical harmonics requires the processing of quadruplets of angular quantum numbers for a given quadruplet of contracted functions. Finally, the generation of symmetry integrals requires the processing of different quadruplets of centers, for a given quadruplet of angular quantum numbers and contracted functions. Clearly, rather sophisticated index manipulations are required. Finally, we have tacitly ignored the use of permutational symmetry: in particular, when the pairs of primitives that form the sets of charge distributions are the same, numerous square blocks can be reduced to triangular arrays of the distinct elements. This also complicates the index manipulations.

CRAY X–MP/48 timing results for some integral calculations on the molecule N_2 are given in Table 8. The basis set used is a ($13s\ 8p\ 6d\ 4f\ 2g$) primitive set contracted to [$5s\ 4p\ 3d\ 2f\ 1g$] using an atomic natural orbital general contraction [51]. The contracted set comprises 140 Cartesian Gaussians, or 110 spherical harmonic basis functions when the contaminants are removed. We shall use this N_2 calculation for timing comparisons throughout this section. In Table 8 it can be seen that the transformation to spherical harmonics requires some 65 seconds on the X–MP/48, but the resulting integral file is less than one-third the length of the Cartesian case. For this N_2 calculation, the saving in time in all later steps as a result of using spherical harmonic functions totals less than 35 seconds, so there is a net increase of CPU time required when spherical harmonics are used. The substantial reduction in external storage required makes the trade-off worthwhile in most circumstances, however.

Table 8 also contains results for the N_2 system treated in lower than the maximum possible (in MOLECULE) D_{2h} symmetry. The C_{2v} group used here has the C_2 axis along the bond, so the two centers are treated as inequivalent. The C_s group is a subgroup of this C_{2v} group. As the symmetry is lowered, the integral time increases, although only slightly on going from D_{2h} to C_{2v}. In this case there is a compensation between a reduction in overhead (as there are no equivalent centers to be generated), and an increase in the number of integrals to be calculated.

A comparison of the performance of the integral generation code on different Cray computers is presented in Table 9. The integrals are generated in somewhat less time on the CRAY–2 than on the X–MP/48: this is rather unusual since for most steps in electronic structure calculations we usually observe the reverse. The improved performance on the CRAY–2 may derive in part from the use of somewhat more memory — 5 MW on the CRAY–2 as opposed to 2 MW on the X–MP — which allows longer vectors to be used in the primitive integral calculation and reduces some overhead. The Y–MP performance for the integral generation is very good: the Y–MP version uses 3.5 MW of memory, so some extra improvement relative to the X–MP is expected beyond the shorter Y–MP clock period. What is observed is actually more than a factor of two, considerably larger than is seen for other codes. No ready explanation for this behavior offers itself.

In addition to the parallelism discussed so far, integral generation lends itself readily to coarse-grained multiprocessing [28]. As each batch of integrals is computed independently, and assuming the integrals can be produced in more or less arbitrary order (as discussed below they will generally need some reordering later anyway), it is possible to compute different batches in parallel. In fact, instead of spawning a new task for each batch it is simpler to divide the range of the four-fold loops over shells in the integral code and execute each subrange as a separate task. With such large granularity the overhead associated with macrotasking becomes an insignificant fraction of the total time, and in a dedicated environment with n CPUs throughput improvements of very close to n times are seen [28].

As will become clear in the succeeding subsections, it is highly desirable to have the integrals ordered on the integral file. This is seldom consistent with efficient integral generation, especially when symmetry is used both to reduce the number of distinct integrals and in obtaining final integrals over symmetry-adapted functions. Consequently, it is usually necessary to reorder the integral file. The reordering of the one-electron integrals is, of course, trivial and we will discuss only the two-electron case. The input/output aspects of this process, especially the use of direct access storage, have been comprehensively reviewed before [52], we will therefore concentrate here on vectorization and discuss input/output only where it has some impact on vectorization.

The MOLECULE program generates four files of two-electron integrals, partitioned according to the symmetry block structure of the integrals. For D_{2h} and its subgroups, there are four types of allowed quadruplets of symmetry species: $\alpha\alpha\alpha\alpha$, $\alpha\beta\alpha\beta$, $\alpha\alpha\beta\beta$ and $\alpha\beta\gamma\delta$, where α etc denote irreducible representations. Each of these types is written to a separate file, as each type can be sorted independently of the others to give the final ordered list. Each raw integral file comprises non-zero integrals together with labels containing the irreducible representations of the four basis functions and their offsets within symmetries. The first step on reading a buffer of integrals is to unpack these labels and identify the position(s) in the final ordered list for the given integral. The label unpacking, which consists of Boolean operations, and the computation of the final position, which consists of multiplications and additions, can all be vectorized over the number of elements in the buffer. The label packing in MOLECULE ensures that the compound indices in the label are strictly non-increasing within a label, so that no conditional structures are necessary in vectorizing the label processing during the read of the raw integral file. Although the calculation of the position addresses involves (slow) integer multiplication and division, X–MP CFT and CFT77 can generate code for performing these operations in the floating-point units (the CFT compiler requires the directive FASTMD [37]). This improves the performance of the loops involved by up to a factor of 20, and of the program as a whole by more than a factor of two. If the number of ordered integrals of a particular type is small enough to fit in main memory (which will often be the case for Cray computer memories and high symmetries) the final position addresses simply represent SCATTER pointer elements into the ordered list, and the raw integrals can therefore be placed in the desired locations with a SCATTER operation, again with a length equal to the number of elements in the buffer.

When the number of ordered integrals of a given type is too large to fit in main memory, a bin sort of the Yoshimine type [52] is employed. The final position indices are used to identify the bins for each integral and its label. Here, because of the possibility that a bin may be full and require emptying before an integral/label pair can be written to it, we encounter a step in the re-ordering that cannot straightforwardly be vectorized. When these bins are read back, however, the only processing required is to use the final position index, offset by the start of the subrange of integrals in a particular chain of bins, as a SCATTER pointer for the integrals. Hence this part of the work is again vectorizable with a length equal to the number of integrals in a bin. On Cray computers this length can usually be several thousand. In fact, since the SCATTER operation will move about 40 MW/s on the X–MP or 20 MW/s on the CRAY–2, it would appear to be necessary to obtain at least the same I/O transfer rates into memory to prevent the re-ordering from becoming I/O bound. With head contention and rotational delays, and disk performance on non-striped mass storage systems, this is hardly possible if the direct access bin file is disk-resident. However, the transfer rate from an X–MP SSD into memory is around 156 MW/s per SSD channel.

(Although some configurations have two SSD channels, it is extremely unlikely in practice that a single user job will simultaneously have access to more than one channel.) Thus if the bin file is entirely SSD resident the re-ordering step will never become I/O bound. With current SSD sizes up to 512 MW and with gigaword (GW) sizes available in the near future this will normally be the case. SSDs are not available for CRAY–2 computers, but for a 256 MW CRAY–2 it will very often be possible to re-order the integrals entirely within memory. Only for large basis sets (upwards of 200 basis functions) and low symmetry will it be necessary to use a bin sort and direct access disk, and so for most calculations the re-ordering will not be I/O bound on the CRAY–2.

For high symmetries (which normally give multi-centered symmetry orbitals) and small molecules the only sparseness in the integral list will come from symmetry. However, for lower symmetries and larger systems (that is for larger distances between basis functions) there may be increasing sparseness in the integral list from the neglect of very small integrals in MOLECULE. It will often be useful to exploit such sparseness to reduce the length of the ordered integral list, as well as the raw integral files. On the other hand, if this is done simply by adding a label word to each integral the sparseness must be greater than a 50% reduction in order to see any reduction in the length of the list. Instead, the index locating a non-zero integral in the full list can be packed into the lowest bits of the integral itself. As each buffer of ordered integrals fixes one pair index ij, it is only necessary to pack the kl pair index, for which it is convenient to use 16 bits. This involves a loss of between four and five decimal places in the mantissa, which is acceptable for most applications, given the 14 digit precision on Cray computers. The examination of the ordered integrals, prior to writing them out, to check whether their magnitude exceeds some threshold, can be viewed as building a GATHER pointer vector (strictly, a COMPRESS pointer): the subsequent GATHER and the packing of the GATHER pointer elements into the lowest bits of the integrals can then be completely vectorized. The entire process can be viewed as a "compressed index" vector operation on the X–MP [11], and under CFT this can be vectorized by the compiler without any need to allocate space for the GATHER index vector. The subsequent processing of such "compressed" integrals is discussed below.

Timing for ordering various integral files is given in Table 8 for N_2 calculations on the X–MP/48. When spherical harmonics are used in D_{2h} symmetry, the ordering can be carried out entirely within 2 MW of memory. This is another useful trade-off against integral generation time when using spherical harmonics: the ordering takes five times longer overall when Cartesians are used, and unless about 3.5 MW of memory is available requires either direct access disk or SSD space. It should be noted that a request for this memory would incur a priority penalty on our X–MP/48, according to the "$1/n$" rule (see section IIID and Ref. 28). When ordering in memory up to 1 000 000 ordered integrals can be produced per second, while for out-of-memory cases the observed best rate depends on the symmetry. In high symmetry cases about 300 000 ordered integrals can be produced in one second, but for the low symmetries this rate rises to more than 500 000 per second. Table 9 contains results for integral ordering obtained on various Cray computers. All of the results given are for sorting in memory. The Y–MP results are somewhat faster than would be expected from the ratio of Y–MP and X–MP clock periods, although the speed increase is not as great as was observed for the integral calculation. The CRAY–2 is considerably slower in this step than the X–MP, because of its much slower memory.

As we shall see, for some applications it is desirable to have a different combination of integrals, such as the "$\mathcal{P}$-supermatrix" with elements

$$\mathcal{P}(ij|kl) = (ij|kl) - 1/4\big[(ik|jl) + (il|jk)\big]. \tag{IV.6}$$

For later convenience the "diagonal" elements $ij = kl$ are usually halved. The reprocessing of the ordered integrals to obtain these values can be handled using very similar techniques to those described above. In fact, as fewer symmetry blocks of $\mathcal{P}$ are usually required than of the integrals themselves, the processing is even simpler.

C. SCF calculations.

The most time-consuming part of most SCF calculations is the contraction of integrals with density matrix elements to build one or more Fock matrices. For the closed-shell Fock operator F this process can be written as

$$F_{ij} = \sum_k \sum_l \mathcal{P}(ij|kl) D_{kl}, \tag{IV.7}$$

where D is the density matrix and $\mathcal{P}$ is the supermatrix introduced in the previous section [53]. Operationally, however, this step is driven by reading blocks of $\mathcal{P}$ from file: blocks corresponding to kl values for fixed ij such that $kl \leq ij$ give rise to two contributions

```
      DO 10 KL = 1,NKL
         F(KL) = F(KL) + D(IJ)*P(KL)
10    CONTINUE
      DO 20 KL = 1,NKL
         F(IJ) = F(IJ) + D(KL)*P(KL)
20    CONTINUE
```

The first of these loops is clearly a SAXPY operation and the second is a dot product. Hence these steps are immediately vectorizable. Further, if the vector P has been compressed to only non-zero values these two steps can be viewed as a sparse SAXPY and a sparse dot product, and are again vectorizable as described in section IIIC above. A more complete discussion of this aspect can be found in Refs. 28 and 54.

For open-shell energy expressions [53,55,56], additional Fock operators must be built from other combinations of integrals, such as the $\mathcal{K}$ supermatrix,

$$\mathcal{K}(ij|kl) = 1/2\big[(ik|jl) + (il|jk)\big], \tag{IV.8}$$

and open-shell density matrices. Such work is obviously vectorizable in complete analogy with the $\mathcal{P}$ processing described above. If multiple open-shell densities are employed, the vectors of K values can be re-used from vector registers, enhancing performance. Alternatively, it is possible to read the $\mathcal{P}$ and $\mathcal{K}$ supermatrices in separate tasks, thereby utilizing more than one CPU simultaneously. It is also possible to use multiple CPUs to process different parts of the same supermatrix, as discussed in Ref. 28. However, if no sparseness is employed the loops for constructing F run at 60 to 100 MFLOPS on the X–MP, and therefore (running the two loops sequentially) require reading up to 25 MW/s from file to avoid becoming I/O bound. Any tactic to increase the effective rate of CPU operations, such as using multiple CPUs, obviously only increases the demands on the I/O system. One possible solution, as discussed previously, is to keep the supermatrix files on the SSD, another, discussed in more detail below, is to keep the files in memory on the CRAY–2.

The only other step in an SCF calculation that consumes anything but trivial amounts of time is the Fock matrix diagonalization, and for most applications with up to 200 basis functions and symmetry blocking this step is quite inexpensive. Although specially optimized matrix diagonalization routines (such as the EISPACK library [57]) are available in CRI's SCILIB library, the requirements for SCF calculations are usually so modest that simple Jacobi or Givens schemes are adequate.

The fraction of the SCF time required for Fock matrix construction is illustrated by the N_2 results of Tables 8 and 9. For the high symmetry calculations 50% or more of the SCF iteration time is used in this step. For the low symmetry calculations the diagonalization of the Fock matrix is relatively more time-consuming. For the D_{2h} case the Y–MP results of Table 9 are essentially what would be expected from its clock period, while the CRAY–2 results are rather poor, given that the sparse dot product and SAXPY performance is expected to be similar to that of the X–MP (see section IIIC). However, the X–MP and Y–MP versions of the SCF code read $\mathcal{P}$ in large fixed-length blocks, so part of the difference between these machines and the CRAY–2 may result from I/O overheads on the latter.

D. Integral Transformation

The treatment of electron correlation requires that the integrals be transformed from the atomic orbital basis set to the molecular orbital basis set. The computational aspects of this step have been extensively reviewed by Wilson [58]. Transformation of integrals can be implicit in some approaches, such as the method of self-consistent electron pairs [59,60], or explicit, involving reading the file containing the AO integrals, transforming these to the MO basis, and writing a new file containing the MO integrals. Hybrid methods for treating the correlation problem are also possible, where some contributions are computed in the MO basis set and some directly from the AO integrals, thus requiring only a partial transformation.

Without symmetry, the full transformation is of the order mn^4, where n is the number of basis functions in the AO basis set and m is the number of MOs. First, it is obviously beneficial to minimize m, and the simplest step in this direction is to eliminate transforming to any orbitals which are not occupied in the correlation treatment. Furthermore, the contribution to the energy arising from electrons which are not correlated (the core) is straightforwardly expressed in terms of Coulomb and exchange integrals only (see, e.g. Ref. 61). This is commonly accounted for by constructing a core Fock operator to be added to the one-electron integrals. In our implementation, this core Fock operator can be constructed using the supermatrix formulation described above for the SCF procedure, or directly from the ordered integrals without further sorting. Clearly, constructing the Fock operator from the supermatrix is faster provided the supermatrix already exists. However, if the supermatrix is not available, the work required to construct one Fock operator directly from the ordered integrals is roughly equal to the time to form the supermatrix — this cost can be amortized over the iterations of an SCF calculation as the supermatrix is used a number of times, but for the core Fock operator construction we prefer to avoid the extra I/O of the sorting step and to construct the operator from the ordered integrals.

In addition to the savings from avoiding transforming for core or deleted virtual orbitals, the overall work can be reduced by exploiting sparseness in the integral or coefficient arrays. While some reduction can be effected simply using sparse vector or matrix arithmetic, in cases where the sparseness derives from the symmetry of the system it is preferable to handle the symmetry explicitly. The use of symmetry in the transformation can reduce the overall work by orders of magnitude: if a basis set of n functions is symmetry adapted with $n/2$ basis functions in each of two irreducible representations, the work is reduced from one transformation of order n^5 to four transformations of $n^5/32$, or an eightfold reduction. Clearly, for systems with higher symmetry, such as D_{2h}, the savings would be even larger. It should be noted that the operation counts given here are based on the assumption that the two-electron integrals are available in a basis of symmetry orbitals. While there are schemes for obtaining the advantages of high symmetry in the SCF step without the formation of symmetry integrals [62-64], such schemes become very complicated for the transformation. However, as described above the symmetry transformation in the calculation of the integrals is very efficient, and the very large savings in the transformation (and SCF) steps that derive from it are real, not a consequence of moving work from the transformation into the integral evaluation. Of course, the simple and efficient symmetry-adaptation procedure described in section IVB is restricted to D_{2h} and its subgroups (although see Ref. 50), but very few treatments of electron correlation are implemented in higher symmetries anyway.

We now discuss our implementation of the transformation, starting with the one-electron integrals. While this requires very little of the overall time, it illustrates several aspects of the transformation of the two-electron integrals. The transformation of the one-electron integrals, **H**, by the coefficient matrix, **C**, can be written as

$$\mathbf{H}' = \mathbf{C}^T\mathbf{H}\mathbf{C}. \tag{IV.9}$$

but the one-electron integrals H_{ij} are symmetric and therefore naturally stored as $i \geq j$. If the integrals are stored only as this lower triangle, it is difficult to vectorize (IV.9). However, if the integrals are expanded to a square matrix (a matrix **T**) the transformation (IV.9) clearly becomes two matrix multiplications. The transformed integrals naturally form a square matrix, which can be compressed to a lower triangle before being written to disk. Based on the timings reported in section III above it is clear that a formulation in terms of matrix multiplications will be very efficient on Cray computers. Hence if the squaring of the AO integral matrix or the compression of the transformed integrals is coded inefficiently, these processes could actually become the rate-limiting step. That is, the pre- and post-processing of the integrals, which are steps of order n^2, could take longer than the n^3 matrix multiplication step, at least for any n that is likely to be encountered in present quantum chemical calculations. Two efficient approaches of squaring (or of compressing) the matrix can be considered. In the first, a row is moved from lower triangular storage to a row and column in the square form.

```
        IX=0
        DO 10 I = 1,N
           DO 20 J = 1,I
              T(J,I)=H(J+IX)
              T(I,J)=H(J+IX)
20         CONTINUE
           IX = IX + I
10      CONTINUE
```

In the second approach a vector of length n^2 is used to hold a mapping into each location in **T** from a location in **H**. Then by using a GATHER the matrix **T** can be easily formed. Similar procedures can be used for the compress. (Note that many GATHER operations can be viewed alternately as SCATTER operations: we use the term GATHER to describe either coding organization). In this approach there is obviously additional overhead in forming the GATHER pointers, but in the transformation of the two-electron integrals the same steps occur so often that the cost of constructing the GATHER pointers once at the beginning of the calculation is effectively amortized to zero.

Either approach to squaring a one-electron integral matrix works well on Cray computers. However, on a machine with a large overhead associated with starting vector operations, like the CDC CYBER 205 or some of the Japanese supercomputers, the GATHER implementation is to be preferred. We have therefore implemented the GATHER approach since it gives us some additional portability.

For large basis sets the two-electron integral transformation poses some problems in data handling. Yoshimine [65] has described an efficient process for transforming two-electron integrals $(pq|rs)$ into $(ij|kl)$ that requires only a small fraction of either $(pq|rs)$ or $(ij|kl)$ to be in core at one time, that is, an out-of-core transformation. In successive steps a list of $(pq|rs)$ is converted into a list of $(ij|rs)$, which is then transposed to a list of $(rs|ij)$ and transformed to a list of $(kl|ij)$. If the ordered integrals are stored such that for each $r \geq s$ all $p \geq q$ values are present (this requires some double storage in the $\alpha\alpha\alpha\alpha$ and $\alpha\beta\alpha\beta$ symmetry blocks) it is possible to read in the "rs row" of the integrals, square the pq indices and perform a two-index transformation on this subset of the integrals analogous to the one-electron integrals. This set of transformed integrals $(rs|ij)$ comprises all ij for fixed rs, and can be compressed to distinct integrals $(i \geq j)$ only before being written to disk. We also compress the final transformed integrals before writing them to file such that only the unique values are stored. That is, any double storage in the $\alpha\alpha\alpha\alpha$ and $\alpha\beta\alpha\beta$ blocks is removed.

Some additional considerations that were not encountered in the one-electron transformation affect the coding of the two-electron case. For example, the number of two-electron integrals can be very large and for extended systems and low symmetries there may be many accidental zeros. As noted in section IVA we have the option to store only those symmetry orbital integrals above a given threshold: each rs row of integrals is compressed to an array of non-zero integrals with the position index of each integral in the uncompressed row packed into its low-order 16 bits, these are written out with the number of non-zero pq values. To expand these values to the full row we clear an array the length of the full row, mask off the low-order bits into an integer array, and use this integer array as the pointer vector in a SCATTER operation. All of these steps are vectorizable, so the process is very efficient. Another aspect of the transformation concerns the appearance of the transposed coefficient matrix $\mathbf{C}^T$ in (IV.9) above. Based on the observations presented in section IIIC, it is better to transpose a matrix explicitly and use the SCILIB routine MXM than to use MXMA to multiply by the transpose, as this minimizes problems with bank conflicts on the CRAY–2. Of course, in the two-electron transformation the same $\mathbf{C}$ and $\mathbf{C}^T$ matrices are used repeatedly, so there is no loss in efficiency (and only a trivial increase in storage) if we store both $\mathbf{C}$ and $\mathbf{C}^T$ and use MXM. Where the integral list is sparse because of accidental zeroes it would be advantageous to use a sparse matrix multiplication routine like Saunders' MXMB [18], but at present we use only the Cray library routines.

While formally the two-electron transformation process can be considered as two series of two-index transformations, another complication relative to the one-electron case is that a transposition of the results is required before the second series of two-index transformations. In our implementation the transposition is merged into the two-index transformations. That is, as each two-index transformation is completed the resulting integrals are moved into bins for the direct access I/O. It is important to note that this process is actually a transposition and not a sort, so there is no index calculation: the first w elements go into bin 1, the second w into bin 2 etc; the value of w is determined by the available memory. This step can be coded as a series of vector moves into the bins. The transposition step is also merged with the second half-transformation: after the contents of a chain of bins is SCATTERed into memory, all related two-index transformations are performed and the fully transformed integrals are written to disk. The process is then repeated for the next chain of half-transformed integral bins.

Another special circumstance in the two-electron transformation arises for symmetry types $\alpha\beta\alpha\beta$ and $\alpha\beta\gamma\delta$, since there is no permutational symmetry between r and s or between p and q in integrals $(pq|rs)$ of these symmetry types. There is therefore no need to square raw integrals or to compress transformed integrals in these cases. However, handling the different cases does not give rise to any noticeable overhead, as we separate the various symmetry types at a very high level (see IVB above). Overall, the inclusion of symmetry involves rather straightforward coding and introduces little overhead, but provides enormous gains in performance.

Two-electron integral transformation timing data for our N_2 example are given in Table 9. The CPU times are effectively negligible in comparison with other steps (notably the integral evaluation) on all machines. As ordered integrals are used as input, and only a transposition is required after the first half-transformed step, the wall clock times are also very small. The CPU times given correspond to a rate of around 50 MFLOPS on the X–MP and CRAY–2, and 65 MFLOPS on the Y–MP. There is actually rather substantial overhead in these times: if a transformation is performed for the C_1 symmetry N_2 calculation described in section IVB the X–MP/48 CPU time required is 324 seconds, but the rate increases to 150 MFLOPS. Thus the use of D_{2h} symmetry in this case gains a factor of 38 in CPU time, but there is a concomitant threefold loss in performance because of the overheads involved in processing numerous rather small symmetry blocks. For a very large basis, in which the transformation overheads would be amortized to nearly zero, the use of D_{2h} symmetry would give an improvement of more than two orders of magnitude over the no symmetry case.

As expected, the out-of-core transformation described here works very well on Cray computers. However, on some other computers, such as the CDC CYBER 205, the large vector start-up overhead leads to poor performance for even the larger matrices. Therefore we have also programmed an in-core transformation. We use a modification of the Bender approach [66]. As in the out-of-core scheme we separate those cases with and without rs permutational symmetry. The first half-transformation is replaced by two matrix multiplications. By ignoring the permutational symmetry between pq and rs, (that is, the $\alpha\alpha\alpha\alpha$ case is treated as $\alpha\alpha\beta\beta$ and the $\alpha\beta\alpha\beta$ case as $\alpha\beta\gamma\delta$), the second half-transformation can be written as long SAXPY operations. The redundant integrals for the $\alpha\alpha\alpha\alpha$ and $\alpha\beta\alpha\beta$ cases are eliminated using a precomputed GATHER pointer vector. This organization yields excellent performance on the CYBER 205. As this approach eliminates the transposition and I/O, it also yields essentially the same performance as the out-of-core approach on all Cray computers, for those calculations for which the transformed integrals can be held in core.

By having both an in- and out-of-core transformation we thus have some additional portability without sacrificing any performance. The program described is so highly vectorized that the n^5 transformation step commonly represents an insignificant fraction of the total time. This is very different from the situation on scalar machines.

E. Full Configuration Interaction

The full CI (FCI) procedure, the use of all configurations that can be constructed from a given one-particle basis set, is an exact solution to the correlation problem for this basis set, and therefore stands as an unambiguous test of approximate methods. Due to recent advances in full CI methodology (principally in vectorization) and developments in computer hardware it has been possible to perform an important series of benchmark calculations [67,68]. Of course, it is still not possible to perform FCI calculations for large basis sets and problems of chemical interest, but it is nevertheless instructive to begin our discussion of correlated wave function generation with the FCI method. Its simplicity makes it ideal for explaining some of the concepts of vectorizing other approaches, and FCI wave functions in a limited one-particle space lie at the heart of the CASSCF method. We therefore consider the FCI approach first and then proceed to the other methods.

In most modern CI calculations, with any type of configuration space, the Hamiltonian matrix is too large to be held in memory, and therefore an iterative diagonalization procedure is required to obtain the CI energy eigenvalue(s) and eigenvector(s). This is true even on the CRAY–2 when configuration spaces of order 10^6 and more are to be handled. The Davidson diagonalization procedure [69,70] is most commonly used. This method has several advantages, one of which is that it imposes no constraints on the order in which matrix elements are processed. This allows the matrix elements to be computed in an order that achieves the maximum overall performance, whereas techniques that implicitly require access to, say, one row of the matrix at a time might lead to intolerable degradation of performance. The key is thus to form efficiently the residual vector σ, the product of the current estimate of the CI eigenvector $\mathbf{c}$ and the Hamiltonian matrix **H**. This step can be written as

$$\sigma_K = \sum_p \sum_q \sum_r \sum_s \sum_L (pq|rs) B^{KL}_{pqrs} c_L, \tag{IV.10}$$

where B are the "coupling coefficients". Siegbahn pointed out [71] that if the known factorization of B^{KL}_{pqrs} into products of one-electron coupling coefficients (using a resolution of the identity),

$$B^{KL}_{pqrs} = \sum_J A^{KJ}_{pq} A^{JL}_{rs}, \tag{IV.11}$$

is explicitly inserted into (IV.10), it is possible to vectorize the calculation of σ very straightforwardly. First, the product of one set of coupling coefficients and **c** is collected in **D**,

$$D^J_{rs} = \sum_L A^{JL}_{rs} c_L. \tag{IV.12}$$

This can be implemented as a set of SAXPY operations involving the elements of the very sparse matrix **A**. A matrix multiplication of the intermediate array **D** and a block of the integrals is performed,

$$E^J_{pq} = \sum_{rs} (pq|rs) D^J_{rs}. \tag{IV.13}$$

The intermediate array **E** is then contracted with the second set of coupling coefficients to form a contribution to σ,

$$\sigma_K = \sum_J \sum_{pq} A^{KJ}_{pq} E^J_{pq}. \tag{IV.14}$$

This operation is a matrix-vector product. It is clear that all steps in the calculation of σ can be vectorized, given that all the necessary quantities are available as needed. This is easily arranged for σ, **c** and the integrals (the latter can be stored in memory for any MO space it is feasible to use in a full CI calculation). The handling of the coupling coefficients **A** requires more attention. As there is a rather large number of A^{KL}_{pq} elements these must either be pre-computed and stored on disk (sorted to an order which allows vectorization of (IV.12) and (IV.14)) or computed on the fly as required in these two equations.

Siegbahn originally introduced this method [71] to improve the performance and increase the size of the FCI step in CASSCF calculations. In this context it was considered appropriate to use spin eigenfunctions as the configuration basis and to compute and store a list of the coupling coefficients on disk. This would require a great deal of disk space for large calculations, but is acceptable for calculations with up to, say, 12 electrons and 12 orbitals in the FCI. For larger calculations, disk space would become excessive, even though the CPU time requirements would remain acceptable, so that disk space becomes the limiting factor. Knowles and Handy [72] showed that by using determinants instead of spin eigenfunctions as the configuration basis the non-zero values of A_{pq}^{KL} (all of which then become ± 1) can be computed on the fly. The CI vector $\mathbf{c}$ is several times longer in a determinantal basis, but this is no handicap on computers such as the CRAY–2. In this way FCI calculations can be performed using very large expansions, providing a means of benchmarking approximate methods as well as performing large CASSCF calculations. The trade-off of memory (and, perhaps, CPU time) for disk storage is, of course, not an uncommon feature of programming modern supercomputers: we discuss below other trade-offs that would not have been considered a few years ago when CPU power was the limiting factor.

As the most time-consuming step in the FCI algorithm (IV.12–14) is a matrix multiplication, the code is very efficient on Cray computers. This has allowed calculations as large 28 000 000 determinants to be performed on the CRAY–2 [73]. Such a calculation requires on the order of 100 minutes of CRAY–2 CPU time per FCI iteration: something over 90% of this time is spent in matrix multiplication. As $\mathbf{c}$, σ and some scratch arrays must be held in core, the memory required for such a calculation is about 60 MW, which is perfectly feasible on the CRAY–2. However, the Davidson diagonalization process requires the $\mathbf{c}$ and σ vectors from the previous iterations. Therefore, just as storing the coupling coefficients can exhaust the disk storage long before the CPU time become prohibitive, so can the need to store the previous $\mathbf{c}$ and σ vectors. As Davidson noted [69], it is possible to "fold" all the previous vectors into one vector, effectively starting again with the current estimate of the eigenvector as the starting guess. While this can slow convergence somewhat, it can reduce considerably the disk storage required. In effect, the CPU performance and the high degree of vectorization, compared to the disk performance and space limits, mandate implementing the folding procedure if the largest possible FCI expansions are to be considered.

In addition to the larger dimension of the FCI problem using a determinantal basis, there is another difficulty: the potential collapse of a desired higher root of the secular problem to a lower root of a different spin symmetry due to numerical rounding [74]. This can be a severe problem in the application to CASSCF wave functions, where many roots of the secular problem may be required in investigating problems related to spectroscopy. Recently, Malmqvist *et al.* [75] have developed a method for using spin eigenfunctions of the desired symmetry (configuration state functions, CSFs) instead of determinants, but without the need to store the coupling coefficients on disk. In their approach the graphical unitary group representation of the CSFs is used (see, e.g. Ref. 76). The graph that defines the configuration space is split into an upper and lower portion, and at each graph node on the dividing line contributions to the product (IV.12) are computed and accumulated using matrix multiplication. This requires subsets of coupling coefficient A_{pq}^{KL} for each portion of the graph, plus some "partial coefficients" for cross-terms between the two portions. All of this coupling coefficient information can be held in memory, even for large configuration spaces (100 000 CSFs and more). Such an approach seems ideal for the CASSCF problem where the number of active orbitals is limited, but may not be suitable for the large calculations used for benchmarking. It is clear that this is currently an active area of research, and as yet the best method of performing FCI calculations may not have been achieved. However, even the current approaches illustrate the very high level of vectorization that can be achieved in CI methods for solving the correlation problem. They also illustrate the need for careful thought about trade-offs in disk storage, memory use and CPU time in order to maximize the size of problem that can be solved.

F. Symbolic formula tape for multireference CI calculations.

The most general type of large-scale CI wave functions in current use is that comprising a set of reference CSFs and all CSFs singly and doubly excited with respect to these reference CSFs. The first task in generating such a wave function is to evaluate the coupling coefficients involved in the various Hamiltonian matrix elements. We begin by classifying the MOs into inactive, active, and secondary orbitals. The inactive orbitals are doubly occupied in all reference

CSFs (but can have other occupations in the CI configuration space — orbitals which are always doubly occupied are absorbed into the effective one-particle Hamiltonian as described in section IVD). The active orbitals have different occupations in different reference CSFs and the secondary orbitals are unoccupied in the reference CSFs. Classes of configurations can be defined by their *internal occupation*, or internal for short. An internal occupation consists of a particular sequence of active and inactive orbitals containing n, $n-1$ or $n-2$ electrons altogether. Those internal occupations that contain n electrons give rise to valence configurations simply by enumerating the possible spin-couplings with which they can appear; valence configurations must have the same spatial symmetry as the desired CI root and must differ from a reference occupation by no more than a double excitation. Internals with $n-1$ or $n-2$ electrons are incomplete, in the sense that to obtain CSFs it is necessary to add one or two secondary orbitals to the occupation. Evidently, a given $n-2$ internal, say, can generate a set of CSFs by coupling in different pairs of secondary orbitals with different spin-couplings. Again, the resulting $n-$electron occupations are constrained to differ from the reference occupations by no more than a double excitation. By expressing a CI configuration list in this manner it becomes clear that the same coupling coefficient applies to a large set of matrix elements; the coupling coefficients are determined essentially by the internals from which the CSFs are derived, so that the particular secondary orbitals that appear are irrelevant. Therefore, if the coupling coefficients are computed for all possible internal-internal interactions, all of the required CI matrix elements can be computed using the appropriate integrals and these coupling coefficients.

In the first direct CI programs the unique coupling coefficients were computed by hand and coded into the program [77]. The types of wave function for which this is feasible are very restricted, and programs following this approach were available only for single reference configuration wave functions based on a closed-shell determinant or a UHF determinant. The direct CI approach became much more general once Siegbahn [78,79] combined the factorization of CSFs (and of coupling coefficients) into internal and external parts with the graphical unitary group approach as formulated by Shavitt [80]. Several variations on this idea were implemented, but most suffered from some limitations in the calculation of the coupling coefficients, a step formulated as essentially a set of scalar operations. Once the coupling coefficients were evaluated, on the other hand, the calculation of the eigenvalues was very efficient, just as in the FCI case, and we discuss this aspect in more detail in the next sub-section. The problem in the evaluation of the coupling coefficients was most severe in the case that the number of references actually used was a small subset of those possible for a given choice of active and inactive spaces, which is unfortunately a rather common case. One alternative approach would be to avoid the unitary group and use a more conventional CI approach. The internal occupations are generated, together with all possible spin-couplings to which they give rise (including coupling of one or two electrons in model external orbitals if required). The occupations are compared to identify potentially non-zero matrix elements, and the coupling coefficients evaluated between prototype CSFs represented by the individual spin-couplings. Such an approach has been implemented [81], but although it can avoid some of the problems associated with the unitary group approach, the calculation of the coupling coefficients is still not vectorizable.

Recent work along quite different lines by Knowles and Werner [82] and by Siegbahn [83] has enormously simplified the calculation of the coupling coefficients. This new approach again exploits the factorization of the two-electron coupling coefficients into sums of products of one-electron coupling coefficients, as in (IV.11). First, the internal occupations are generated. In computing the one-electron coupling coefficients a product form is used: a fictitious MO a is introduced which is unoccupied in all the internals, whereupon we can write

$$A_{pq}^{KL} = \sum_{M} A_{pa}^{KM} A_{aq}^{ML}, \tag{IV.15}$$

where M is an "internal" to which a has been coupled. The A_{pa}^{KM} values are very simple to calculate and the values can be held in memory. In evaluating the two-electron coupling coefficients, pairs of internals are compared and the type of coupling coefficients that arise are identified (these types are determined by the orbital differences between the pair and the open shells they contain). The coupling coefficients between all CSFs that can be derived from the pair of internals are then evaluated by matrix multiplication of the stored one-electron terms. The corresponding formula tape entry comprises the internal labels, the label of the integrals (or blocks of integrals) that appear, and the coupling coefficients. In this approach there is little redundant work even where only a few of the possible reference occupations arising from a given active space are used. Furthermore, unlike the approach using prototype CSFs described above, the evaluation of the coupling coefficients is highly vectorized. The program that we use was developed by Siegbahn [83] and performance better than 50 MFLOPS has been observed on the X–MP/48. Some timing results are given in Table 10. For our N_2 example the coupling coefficients for single and double excitations from a single closed-shell reference CSF (16 internal occupations) requires much less than one CPU second, while for the case of a CAS reference space that gives rise to 2804 internals the coupling coefficients require 12.5 seconds. These very recent developments means that the rate-limiting step in an MRCI calculation is in the calculation of the eigenvector, and the excellent performance of the new approach allows very large reference spaces: the calculation of the coupling coefficients for more than 60 000 internals requires about 700 seconds on the X–MP/48.

G. Multireference CI eigenvalue determination

The FCI calculations described above show that it is possible to achieve very high performance in the eigenvalue determination through the formulation of the time-consuming step in terms of matrix multiplication. The factorization into internal and external contributions in the multireference direct CI (MRCI) case, with the consequent association of a set of CSFs and an array of CI coefficients with each internal occupation, is also very well suited to a matrix multiplication formulation [81,84-86], and so again very high performance should be possible. On the other hand, there are substantial differences between the FCI and MRCI cases, resulting mainly from the fact that all orbitals in the FCI calculation are classified as part of the same orbital space and their number is so small that all integrals can be held in memory, while in the MRCI calculations there is a distinction between occupied and secondary orbitals, and it is seldom possible to hold all the integrals in memory. For MRCI calculations, additional data-handling involving sorting both integrals and coupling coefficients must thus be performed. If $i, j, k...$ are active or inactive orbitals and $a, b, c...$ are secondary orbitals, the types of interactions between internal occupations can be classified by the type of integrals required: $(i|h|j)$, $(a|h|i)$, $(a|h|b)$, $(ij|kl)$, $(ai|jk)$, $(ab|ij)$, $(ai|bj)$, $(ab|ci)$, and $(ab|cd)$. Each class of integrals contributes to a limited number of types of interaction. Therefore the determination of the eigenvalue first involves sorting the integrals into classes, and to a given order within classes. The formula file is sorted during its generation to have the same organization of types as the integral file. In each MRCI iteration the program loops over each type of interaction and within each type a block of coupling coefficients is read. If these coupling coefficients refer to a new block of integrals, these are also read in. In our organization, the formulas and integrals are ordered so that the integral and formula files are read only once per iteration, and of the order of N_{MO}^2 integrals are required to be in memory at one time, where N_{MO} is the number of MOs.

Each possible interaction is treated in a subroutine specific to the integral class: the details of the treatment have been extensively reviewed by Saunders and van Lenthe [86], and we will not repeat them here. As shown in Ref. 86, most types of interaction can be reduced to multiplying a block of integrals by coupling coefficients, and then a subsequent matrix multiplication of the results by a block of CI coefficients. (In practice, the block of integrals may actually be a sum of two different blocks multiplied by two coupling coefficients.) The final matrix product is added to a block of the σ vector, so there is some saving of memory if the matrix multiplication routine allows the product to be added to (or subtracted from) the destination array.

Symmetry is used in all sections of the CI code, just as in the transformation. This includes the index permutation symmetry of the integrals as well as the spatial symmetry. If P indexes a particular $n-2$-electron internal, for example, the coefficients of the distinct double excitations generated by coupling to P can be written as an array

$$c_P^{ab} \quad \forall \;\; a \geq \; b. \tag{IV.16}$$

Such arrays display the same permutational symmetry as one-electron integrals: if a and b are of the same symmetry type the array is triangular, while if they are of different symmetries the array will be square. Some contributions to σ will then be required only as lower triangles, as the blocks of σ have the same structure as those of $\mathbf{c}$. In such cases we must square the block of integrals and CI coefficients, in the same manner as in the transformation, form the necessary product and then fold the two halves of the product matrix together to add to the σ vector. These squaring and folding operations are vectorizable, as discussed in IVD above, and take only a small amount of time. Therefore, while an MRCI calculation involves a far more complex organization than the FCI scheme discussed in IVE, each individual step is vectorizable and very high performance can be obtained. We have been able to routinely perform calculations involving 1 million CSFs and have on occasion performed calculations involving more than 4 million CSFs.

Some timing results are presented in Table 10. For N_2 with the 10 valence electrons correlated and an SCF reference the direct CI iteration time is less than 1 second on the X–MP/48. This calculation involves 17626 CSFs. The observed performance is about 44 MFLOPS: lowering the symmetry to C_{2v} doubles the iteration time but also improves the performance to 66 MFLOPS. In the limit of no symmetry about 140 MFLOPS would be obtained. Hence for very large basis sets, in which the overheads have become negligible, we may expect single reference direct CI performance of about 140 MFLOPS on the X–MP/48. In multireference cases, the performance is somewhat higher. For example, timing for an N_2 calculation with a CAS reference space (six active electrons in six active orbitals, giving 32 reference CSFs) is also given in Table 10. The iteration time of about 80 seconds for almost 730 000 CSFs corresponds to an average of 50 MFLOPS. Hence in very large (or low symmetry) cases we may expect MRCI performance of better than 150 MFLOPS. A breakdown of the that part of the iteration time concerned with the three most time-consuming classes of integrals processed is also given in Table 10. The $(ai|bj)$ integrals (this actually includes $(ab|ij)$ integrals as well) require the most effort, followed in this case by $(ai|jk)$ and $(ab|cd)$. For very large basis sets the latter are expected to dominate, but this case is rarely observed in practice. Finally, we note that on the CRAY–2 the observed times are up to twice as long as on the X–MP/48: these N_2 calculations involve multiplication of rather small matrices and the matrix multiplication performance on the CRAY–2 will suffer somewhat. On the other hand, the iteration times on the Y–MP are about 2/3 of those on the X–MP, suggesting that large MRCI calculations will run at well over 200 MFLOPS on the Y–MP.

H. CASSCF calculations

The CASSCF approach can account for near-degeneracy correlation effects and correlation contributions that vary rapidly with geometry [87]. It can thus supply an excellent zeroth-order description for use in the MRCI approach.

We employ a two-step, uncoupled MCSCF optimization scheme [88], which involves a transformation of the integrals, determination of an FCI wave function, construction of a gradient vector and Hessian matrix for the energy dependence on orbital rotations, and solution of the simultaneous linear equations of the Newton-Raphson method for obtaining improved orbitals. In practice, far from convergence it is better to use a first-order approach for optimizing the orbitals [89,90], in which case an approximate Hamiltonian matrix over single excitations must be constructed instead of the Hessian; this Hamiltonian matrix is then diagonalized in order to obtain improved orbitals. We now consider each of these steps in more detail.

Just as for the MRCI case discussed in the previous subsection, different classes of integrals are required in a CASSCF calculation. For the CI step, only integrals with four active orbital indices are required (the inactive orbital contributions can be absorbed into the one-electron operator, as discussed in IVD above), while for the gradient of the energy with respect to orbital rotations, integrals with three active orbital indices and one index that runs over the full MO space are required. Finally, for the orbital rotation Hessian, integrals with two indices running over the full space and two over the space of occupied (inactive and active) orbitals are required. We compute all integrals required for a given MCSCF iteration in the same transformation step. The two-electron integral transformation is performed in a manner very similar to the full two-electron transformation described above in IVD, but there are some special features added because only certain classes of transformed integrals are required. For example, if $m, n...$ denote occupied MOs and $p, q...$ all MOs, it is clearly advantageous to obtain integrals $(pq|mn)$ and $(pm|qn)$ by first transforming integrals over index n and then over q. In this way no step in the transformation scales worse than MN^4, for N AOs and M occupied MOs, an N/M-fold reduction over the full transformation. Further, the ranges used in the second half-transformation can obviously be modified depending on which MO indices appear in the half-transformed integrals. Some care is needed in this approach when symmetry is used. Thus if the integrals $(p_\alpha q_\alpha|m_\beta n_\beta)$ are required, where α and β label irreducible representations, it would be necessary to perform a full transformation on the α symmetry indices at the outset if the $\alpha\alpha\beta\beta$ AO integral block is to be processed only once. Instead, following Roos [91], this block is processed twice in the first half-transformation, once as $\alpha\alpha\beta\beta$ and once as $\beta\beta\alpha\alpha$. In this way the first transformation step always scales as MN^4. The same strategy is employed for the $\alpha\beta\gamma\delta$ blocks as well.

The FCI calculation is handled using the factorized coupling coefficient approach of Siegbahn [71], as described in section IVE. The construction of the orbital Hessian requires a very small amount of time and in our current version is still scalar code. The solution of the simultaneous equations is performed using an iterative scheme analogous to the Davidson diagonalization [69]. As we are normally able to store the full orbital Hessian in memory, the matrix-vector product needed in the iterations can be performed one row at a time, it is then identical to the construction of the Fock matrix (eqn IV.7 above) and is implemented as a SAXPY and a dot product. (The handling of the approximate Hamiltonian matrix appearing in the first-order optimization scheme is very similar.) Note that for very large cases (or in the absence of symmetry) this step could also proceed as for the Fock matrix construction by storing the ordered rows on disk and re-reading them in each iteration. If necessary, the same compression techniques as used for the SCF supermatrices could be employed, together with a sparse SAXPY and dot product.

The time-consuming steps in a CASSCF calculation are thus vectorizable, and using these techniques we have been able to perform calculations involving about 40 000 CSFs and a basis set of about 200 functions. For the N_2 example we have used in these discussions the CASSCF iteration time is dominated by the integral transformation (which takes only a few seconds on the X–MP/48), the orbital optimization step requires one second and the CI time is negligible (and is anyway almost entirely overhead for this case — 32 CSFs). For a configuration space of some 3500 CSFs the CI step requires about five seconds per direct CI iteration, while spaces of 40 000 CSFs would require about 80 seconds on the X–MP.

I. Avoiding I/O, direct methods

One of the constraints on how efficiently calculations can be performed, as we have repeatedly noted, is the rate at which data can be retrieved from secondary storage. SSDs provide a partial answer to this problem for X–MP (and Y–MP) machines, but these devices are not available for the CRAY–2. However, the large central memory provided on the CRAY–2 and planned for the CRAY–3 provides a possible alternative: that of memory-resident data sets. One obvious technique is to generate the integral list in memory for use in subsequent SCF steps. With the use of high symmetries (generating only symmetry-distinct integrals) and pre-screening techniques to eliminate small integral batches, it is possible to perform calculations with 500 basis functions or more in 150 MW or less on the CRAY–2 [28,92]. It is advantageous to avoid storing labels with the integrals by processing the list using the same loop structure as was used to generate them: in the first "SCF iteration" the non-vanishing distinct integrals are computed, stored and used in the Fock matrix build, and a flag is set in an index list to indicate that a particular batch was computed; in subsequent iterations the same loops are executed, skipping all processing if the batch was not computed and simply retrieving the batch from memory for the Fock matrix build if it was. While there is some overhead associated with the repeated execution of the outer loops of the integral generation code, considerable storage is saved by eliminating labels. As the flag for each batch can be represented by a single bit the index list requires almost no space. The whole scheme requires little more CPU time than a conventional SCF scheme, but eliminates almost all I/O processing. This approach has also been used to reduce the I/O associated with the perturbed SCF and CASSCF wave function generation in the ABACUS analytic derivative program [93,94].

A more drastic approach to the elimination of I/O is not to store large data sets (such as the AO integrals) but to recompute them as required. This is the philosophy behind the "direct SCF" scheme of Almlöf and co-workers [47,95]. Viewed naively, this appears to represent a trade-off between the CPU time required to generate the integrals and the I/O overhead (and storage requirements) associated with repeatedly retrieving them from secondary storage. However, it is perfectly possible for machines like the X–MP or CRAY–2 to generate the integrals over basis sets of 1500 functions or more in reasonable CPU times (say, on the order of hours); that is, to generate an integral list far larger than the capacity of most supercomputer secondary storage systems. In such a case the question of a trade-off does not even arise, as the conventional approach would not be feasible. Of course, since the integrals must be regenerated in each iteration, it is desirable to minimize the number of iterations and to eliminate as many integrals as possible, as well as using the most efficient scheme possible for computing the integrals. These aspects, together with timings, have been discussed elsewhere [47,95,96], as has an interesting hybrid approach with explicit storage of some integrals [97].

The extension of this direct approach to MCSCF and CI calculations has also been discussed, but only in purely formal terms without computational implementation [98]. Very recently, however, Saebø and Almlöf [99] have developed a program for computing the MP2 correlation energy using a direct approach. Such a scheme can be regarded as a first step towards implementing a CI method, as well as generating results which are very useful in themselves. The MP2 energy requires MO integrals of the form $(ia|jb)$, which are most efficiently obtained not by transforming the charge distributions, as described above, but by transforming the first and third indices as the first pair, then the second and fourth. It is obvious that this requires an integral list that is four times longer than the "canonical" list, that is, double the length required for the conventional transformation. In a direct approach to MP2, then, about four times as many integrals must be calculated as in an SCF iteration, so it would be expected that the MP2 energy would require about four times the CPU time of a direct SCF iteration, assuming that integral generation is the dominant step. This is essentially what is observed in practice by Saebø and Almlöf [99]. A similar approach has also been investigated by Head-Gordon *et al.* [100].

J. The influence of supercomputers on quantum chemistry.

It is important to conclude this presentation of supercomputer implementations with a discussion on how these algorithms and machines have influenced our approach to quantum chemistry. While it might be thought that performance gains of an order of magnitude or more would simply increase the size of systems considered, their influence is much more profound. For example, the ability to perform full CI calculations in realistic basis sets has provided us with detailed benchmarks for other correlation methods [67,68]. These have shown that multireference CI wave functions (with some correction for size-consistency effects if eight or more electrons are correlated) reproduce the full CI results to very high accuracy. Since MRCI wave functions are thus an adequate solution to the correlation problem, we may infer that (assuming relativistic effects or Born-

Oppenheimer breakdown terms are negligible) any discrepancies between MRCI results and experimental observations are due to inadequacies in the atomic orbital basis. While this had been hypothesized on a number of occasions [101-103], full CI calculations (being more practical than the use of a complete orbital basis) were required to confirm it. It then becomes worthwhile exploring the possibility of using better basis sets together with MRCI wave functions, and supercomputers provide the necessary computational resources to allow the use of atomic natural orbital basis sets [51], which have led to almost chemical accuracy (1 kcal/mol) for systems such as N_2 [104], and to even higher accuracy for CH_2 and SiH_2 [105].

Another way in which the power of supercomputers influences quantum chemistry is the speed with which results can be obtained. Even if results of a desired accuracy can be obtained on, say, a VAX-type minicomputer, the real time required to obtain the results may be too long for the calculation to be useful. It is an important consequence of using a supercomputer that results can be obtained in relatively short times, and this factor will doubtless increase in importance as supercomputer performance increases.

V. Dynamics

A. General observations

While much of the effort in computational quantum chemistry goes into implementing a few, rather well explored methods, this is much less true in scattering calculations. It is generally necessary to consider a wider range of possible methods and their various implementations, and the optimum course is often much more application dependent. We shall review here a greater variety of different methodologies than was the case for quantum chemistry, but there is still no intention to review the entire field; we concentrate on approaches that we have explored and used.

B. Classical dynamics

The simulation of collisions by the quasi-classical trajectory method can be broken down into three steps: the specification of the initial conditions for the trajectories, the integration of the equations of motion to determine the final conditions, and the analysis of the the final conditions to extract rate data [4,5]. The initial coordinates and momenta are determined from quantities which fall into two categories, namely those which are fixed, like the total energy or initial quantum state, and quantities such as orientations, which are not experimentally resolved. The unresolved quantities must be averaged over, and this necessitates the determination of many different trajectories.

Computationally, the most expensive step is the integration of the trajectories. It is therefore advantageous to consider the savings possible by vectorizing this step. The problem is to determine the coordinates q_i and their conjugate momenta p_i at some time after the collision, given values before the collision. These are determined by solving Hamilton's equations:

$$dp_i/dt = -\partial H/\partial q_i, \tag{V.1}$$

and

$$dq_i/dt = \partial H/\partial p_i. \tag{V.2}$$

Here H is the Hamiltonian and i runs from 1 to the number of degrees of freedom in the system. Thus once the center of mass motion is removed, Hamilton's equations comprise a set of $6N-6$ coupled first-order differential equations, where N

is the number of atoms. In the absence of external fields, the derivatives of the Hamiltonian become $\partial H/\partial q_i = \partial V/\partial q_i$ and $\partial H/\partial p_i = \partial T/\partial p_i$, where V is the potential energy of the system and T is the kinetic energy.

Many different algorithms have been used to solve Hamilton's equations, however in most methods the time-consuming steps consist of forming the quantities $\vec{f}_j$, the vector of time derivatives of the p_i and q_i at intermediate step j, and then using them to predict new coordinates and momenta. For later convenience let $\vec{y}_n$ be the vector of p_i and q_i at time step n. The operations for predicting new coordinates and momenta are typically SAXPY-like operations which can be evaluated reasonably efficiently, as demonstrated in previous sections, provided the vector lengths are great enough. If a single trajectory is being integrated, then the vector lengths would be $6N - 6$, or 12 for A+BC collisions and 18 for AB + CD collisions. These lengths are insufficient to give execution rates which approach the ultimate capabilities of vector pipelined machines. Further, the vector lengths involved in the calculation of the gradients of the potential which contributes to $\vec{f}_j$ will be even less. Vectorizing a single trajectory is thus not a very efficient use of Cray computers.

It is fortunate that it is possible to do much better by taking advantage of the fact that many trajectories are required. Because the integration of each trajectory is independent of all others, several can be processed simultaneously. That is, new vectors $\vec{Y}_n$ and $\vec{F}_j$ can be constructed by simply concatenating the vectors $\vec{y}_n$ and $\vec{f}_j$ from different trajectories; these new longer vectors can then be used in the predictions of new coordinates and momenta. The vector lengths in these steps can thus be made equal to $(6N - 6)N_{traj}$, where N_{traj} is the number of trajectories integrated simultaneously. This means that with little difficulty the asymptotic rates can be reached for the operations that are performed. However, the integration is not the entire calculation, and several other steps need to be considered before predictions of overall execution rate can be made.

We note first that although the integration of each trajectory is independent of the others, the initial conditions are not. This is because the random sampling used to generate the initial conditions relies on pseudo-random number generators which require knowledge of one or more previous pseudo-random numbers in order to generate the next in the sequence. Thus in general this initialization part of the calculation must be performed in scalar mode. Fortunately, it is of sufficiently small size that the scalar operations do not contribute significantly to the overall run time.

An additional possible complication is that if variable step size algorithms are used to integrate Hamilton's equations, then operations which are more dependent on the specific trajectory are required to predict the coordinates and momenta. This would disrupt the vector processing discussed above. Consequently it is necessary either to use a fixed step size algorithm, or to modify existing variable step size methods to treat the different trajectories in a more uniform manner. The optimum solution will be problem dependent, and in many cases the simplicity of a fixed step size algorithm will out-weigh the advantages of a variable step size algorithm.

The final aspect to consider is the evaluation of the time derivatives $\vec{f}_j$, that is, the right hand sides of (V.1) and (V.2). For systems for which cartesian coordinates are used as the q_i, the derivatives of T are simply mass factors times the p_i and thus are relatively trivial to form compared with the other parts of the calculation. Thus the calculation of dp_i/dt and dq_i/dt, given values of p_i and q_i is mainly spent evaluating the quantities $\partial V/\partial q_i$. This evaluation can be further broken down into two steps. First of all, it is unlikely that the potential function will be known explicitly in terms of the integration coordinates q_i: typically the interatomic distances might be used to express the potential but Cartesian coordinates for the q_i. Thus it is necessary to first compute the gradients with respect to some set of internal coordinates Q_n, and then to use the chain rule to obtain the final derivatives. Usually the manipulations required for the chain-rule calculations are straightforward and inexpensive compared to the last remaining step, the evaluation of the gradient of the potential. For systems in which a realistic potential is used, this step will usually dominate all other steps. This is simply a manifestation of the complexity of a typical expression for the gradients of the potential compared to the expressions for the integration algorithms.

For relatively simple potentials, it can be advantageous to optimize the various operations taken by the integrator more carefully — see Ref. 106 for an example of how this can be done.

The operations required to generate the gradient of the potential depend on the representation of the potential: typically, mathematical functions like exponential and square root, and various trigonometric functions are required, as well as the usual arithmetic operations. In addition, functions requiring table lookups, such as splines, are sometimes utilized, thus special effort is required to produce an efficient code. However, in contrast to procedures which rely heavily on matrix multiplication, the maximum execution rate which is ultimately achievable is usually well below the theoretical hardware limits. This arises for two reasons. First, the expressions for the gradients often contain common factors which are the result of single operations: these cannot be chained with other operations and such steps will produce a maximum of a single result per clock period. Second, the possibilities for minimizing memory traffic by unrolling loops or by using several vectors (which remain in the vector registers) for more than one operation are limited. This is again due to the complexity of the expressions for the gradients and the many different vectors involved. Thus while all of the time-consuming steps for classical dynamics calculations can be vectorized, the vector speedups obtainable are limited because of the relatively inefficient mapping to the hardware.

Perhaps the best way to improve the performance of the construction of the gradients of the potential is simply to minimize the number of evaluations required that is, to use an efficient variable step size integrator. However, this can cause inefficiencies elsewhere, because of vectorization difficulties, and the various parts of the calculation must be judiciously balanced for overall efficiency. One example where the extra work involved in a variable step size algorithm paid off was in calculations on the recombination of hydrogen atoms [107]. Here one H_2 molecule will have an energy near or above the dissociation limit, and at certain times this energy can be mostly kinetic energy, in which case small time steps will be required, while at other times the energy will be mostly potential energy and the slow velocities would allow larger time steps. Since most of the time will be spent in regions of configuration space where the energy of the diatomic is mostly potential energy, a variable step size algorithm can provide significant savings over a fixed step size algorithm. The variable step size algorithm implemented for this problem was a modification of the Bulirsch-Stoer method [108]. This method has been shown to be an efficient choice for the scalar integration of trajectories [4], and has the additional advantage that minor modifications allow the vectorization of most of the operations involved in the integration. We only outline the ideas here: full details are given in Ref. 107. The basic philosophy of the method is that to propagate the solution over some time increment, a series of integrations using a low-order method and smaller and smaller step sizes is performed. The results of each individual integration will not necessarily be very accurate, but accurate results can be obtained by extrapolating the results of the different step sizes. Based on the number of integrations required to obtain an accurate extrapolation, the time increment to be used next is predicted, that is, the step size is adjusted. In typical scalar implementations of this method, the number of low-order integrations per time increment is increased until the extrapolation converges within some tolerance. However, a similar scheme would not be very feasible for the integration of several trajectories simultaneously. Thus the method was modified to perform a fixed number of low-order integrations per time increment. Then, based on the errors in the extrapolations using the different numbers of integrations, the results for that time increment are either discarded and the time increment for that trajectory decreased, or the results are saved and the time increment adjusted to reach some error goal. Thus the scalar operations for the variable step size part of the code consist only of error checking, saving or discarding the results and adjusting the time increment. These steps amount to such a small fraction of the overall process that this algorithm performs very efficiently on the Cray machines.

As an illustration of the discussion above, we offer the times given in Table 11. Here we show timings using the variable step size Bulirsch-Stoer integrator for two systems. The first system is F + H_2 using the simple Muckerman No. 5 LEPS potential [109]. This system was chosen to give an idea of the limit of a three-body system with a simple potential. The second system is H_2 + H_2 using the accurate potential from Ref. 107 which is based on extensive *ab initio* electronic structure calculations, and represents the extreme case of a complicated potential for a system with many degrees of freedom. Both calculations use interatomic distances as the internal coordinates Q_n with respect to which the potential gradients are directly calculated, and Cartesian coordinates having the center of mass stationary at the origin as integration coordinates [107]. The times are normalized so that the integration time is 1 unit on the X–MP/48, and the MFLOPS rates are determined from the hardware performance monitor on the X–MP/48 and by scaling by the appropriate times for the other machines. The number of simultaneously integrated trajectories (N_{traj}) was 500.

First, consider the F + H_2 system. Even when using the very simple potential employed here, a considerable fraction of the total time is spent evaluating the gradients of the potential with respect to the integration coordinates. The amount of time spent in the integration routine amounts to only 36% to 51%, depending on the machine. Thus in spite of the modest execution rate of the variable step size part of the code, a reasonable overall rate is achieved. Of the time spent getting the gradients, about 20–27% is spent calculating the interatomic distances and transforming the gradients to the integration coordinates. Consider now the H_2 + H_2 system. Here the complexity of the potential is obvious — 94% of the time is spent evaluating the gradients of the potential with respect to the internal coordinates in spite of the fast execution rates of 100 – 190 MFLOPS. Here since only 3% of the time is spent in the integrator, considerable flexibility is available in optimizing the variable step part of the code by introducing more scalar operations, without significantly degrading performance. The relative CPU time per integration step is about a factor of 20 greater for H_2 + H_2 than for F + H_2.

The overall execution rates for either of the two potentials on the various machines range from 100 to 190 MFLOPS, with the Y–MP at the top of this range and all the other machines clustered at the bottom. These rates are well below the theoretical hardware limits, which is probably a reflection of the limited optimization ability of current compilers when faced with the complicated expressions present in these codes.

At this juncture, we discuss other resources required by the classical dynamics calculations, namely disk and memory. As a rule, the variables required for the various operations required to propagate a trajectory are of sufficiently small number that they all can be held in memory, thus very few I/O operations are required during the integration of a trajectory. An exception to this is if the coordinates and momenta along the trajectories are desired for analysis purposes, such as plotting. Then it will be necessary to perform a certain amount of I/O in order to save these quantities for later use. However, the majority of trajectories will not be so analyzed, and thus it is only necessary to save the initial and final conditions. The memory allocated by our program has not been minimized, mainly because we have not encountered problems obtaining the space desired. Much of the space is taken up by temporary vectors, which a more sophisticated compiler could allocate dynamically, thus reducing the overall space requirements. The present code used for $H_2 + H_2$ requires $720 \times N_{traj}$ words for the integration part of the code and $589 \times N_{traj}$ words for the potential gradient part of the code. Thus when using $N_{traj} = 500$, which is our usual production value, the code requires less than one million words of memory.

This section will be closed with some ideas on how the timings for trajectories could be improved. Since most of the time is spent on evaluating the gradients of the potential, we will concentrate on this aspect of the problem. With the recent availability of *ab initio* gradient methods, it would seem advantageous to directly fit the gradients of the potential. Then one would be evaluating different functions for the various gradients rather than differentiating a single function. Although this procedure has many conceptual merits, it has several possible problems that limit its usefulness. First of all, the fitted gradients will probably not all integrate to the same function, that is, the potential will be nonconservative [110] and so the trajectories will not conserve energy. However, in practice this may not be a significant problem. Another problem is that often both the potential and the gradients are required. For instance, it is often desirable to calibrate classical methods by comparing to quantum mechanical solutions of the problem, and quantum mechanical dynamics methods require the potential itself rather than the gradients. In addition, hybrid dynamical methods exist [111] which treat some degrees of freedom using classical mechanics and some using quantum mechanics. Another facet of this reflects the large amount of labour usually involved in constructing a fit to the potential: it not uncommon for several different studies using different methods to be performed using a potential once it has been fitted. It is thus less restrictive if the potential is given rather than the gradients. Finally it may simply not be as efficient to evaluate separate fits to the gradients as it is to explicitly differentiate a function. This is because there may be many common expressions in the differentiation formulas. For example, if we make the assumption that the gradients will be fitted by a function of similar complexity as the potential, then a comparison of the number of gradients times the time to evaluate the potential to the time to evaluate the potential gradients will give some idea of the efficiency of the method. For the simple $F + H_2$ potential used in the timings above, the time to generate the potential and three gradients is only 1.3 times as long as just generating the potential, and for the H_2 + H_2 potential of Ref. 107, the time to generate the potential and six gradients is only 2.3 times the time for just generating the potential alone. Thus for these cases, it is more efficient to deal with the potential and then explicitly differentiate it.

B. Quantum dynamics

In the differential equation approach to quantum mechanical dynamics calculations of nonreactive molecular collisions, the equations to be solved are [112]

$$\frac{d^2}{dr^2}\mathbf{f}(r) - \mathbf{D}(r)\mathbf{f}(r) = 0, \tag{V.3}$$

where r is the distance between the centers of mass of the target and projectile, the $\mathbf{f}$ are the unknown radial functions and $\mathbf{D}$ is the coupling matrix. The matrix elements of the coupling matrix are given by

$$D_{nn'} = \frac{2\mu}{\hbar^2} < n|V^{int}|n' > +\delta_{nn'}[-k_n^2 + l_n(l_n+1)/r^2], \tag{V.4}$$

where $< n|V^{int}|n' >$ is the matrix element of the interaction potential between the basis functions labeled by the indices n and n', μ is the reduced mass for the collision partners, k_n^2 is the square of the wave vector for channel n:

$$k_n^2 = 2\mu(E - \epsilon_n)/\hbar^2, \tag{V.5}$$

E is the total energy, ϵ_n is the internal energy for channel n, and l_n is the orbital angular momentum quantum number for relative translational motion for channel n. The basis functions labeled by n describe all degrees of freedom except r and each value of n defines what is known as a channel. The interaction potential is defined as that part of the potential which goes to zero as r goes to infinity, thus it does not include the binding potential of the collision fragments.

The boundary conditions for the unknown functions are

$$f_{nn'}(0) = 0, \tag{V.6a}$$

and as r goes to ∞,

$$f_{nn'} \sim \begin{cases} (2ik_n)^{-\frac{1}{2}}\{\delta_{nn'}\exp[-i(k_nr - l_n\frac{\pi}{2})] - S_{nn'}\exp[i(k_nr - l_n\frac{\pi}{2})]\}, & k_n^2 > 0; \\ i(2|k_n|)^{-\frac{1}{2}}\{\delta_{nn'}\exp[|k_n|r] - \tilde{S}_{nn'}\exp[-|k_n|r]\}, & k_n^2 < 0. \end{cases} \tag{V.6b}$$

From the matrix elements of the scattering matrix, $S_{nn'}$, experimental observables can be calculated using standard formulas [113-115].

To determine the scattering matrix, we convert the problem of determining the $\mathbf{f}$ from a boundary value problem to an initial value problem by determining a linearly independent set of solutions of (V.3), called $\tilde{\mathbf{f}}$ which are regular at the origin but have arbitrary slopes there [116]. In practice it is not usually necessary to start at the origin but rather at any larger distance where the hard core of the potential is sufficiently repulsive so that the radial wave functions are negligibly different from zero. At large r these functions behave as

$$\tilde{f}_{nn'} \sim \begin{cases} P_{nn'} \sin(k_n r - l_n \frac{\pi}{2}) + Q_{nn'} \cos(k_n r - l_n \frac{\pi}{2}), & k_n^2 > 0; \\ \tilde{P}_{nn'} \exp[|k_n|r] + \tilde{Q}_{nn'} \exp[-|k_n|r], & k_n^2 < 0, \end{cases} \tag{V.7}$$

with $\mathbf{P}$, $\tilde{\mathbf{P}}$, $\mathbf{Q}$, and $\tilde{\mathbf{Q}}$ determined from $\tilde{\mathbf{f}}$ at two points or the logarithmic derivative at one point. From $\mathbf{P}$ and $\mathbf{Q}$, the scattering matrix can be easily determined.

The number of algorithms for solving the close-coupling equations (V.3) is quite large [117-121], and the optimum algorithm is case dependent. The number of coupled equations, the dimension of $\mathbf{D}$ in (V.3), will be called N and can range from 1 for central potential problems to over a thousand for anisotropic AB + CD collisions [122], thus the vectorization strategies will depend on the size of the problem. In contrast to problems in classical mechanics, the integration of (V.3) for a single channel ($N = 1$) can be vectorized relatively effectively. This is because the interaction potential depends only on the parameter r, which can be made to take on known values while in classical problems the potential depends on the unknown functions which are being determined. Thus the steps required for integrating the close-coupling equations for a single channel typically would be to determine $D_{11}(r)$, to determine temporary quantities depending on $D_{11}(r)$ for single values of r, and finally to recursively assemble the solution. The first two steps are completely vectorizable, with vector components corresponding to different values of r, while the final step is not; however, if properly coded this final step will not involve many operations per r value and hence will not be the rate-determining step. For example, in Ref. 123, scattering calculations were carried out at complex energy searching for poles in the scattering matrix. This required calculations at many different energies. The algorithm used was a modification of the R–matrix propagation algorithm [124-126] where the equations defining the modified method are

$$\tilde{R}_4^{(1)} = 0, \tag{V.8}$$

$$\tilde{R}_4^{(i)} = t^{(i)}/(\tilde{R}_4^{(i-1)} + q^{(i)}), \tag{V.9}$$

$$f_{11}/(df_{11}/dr|_{r=r_i+h_i/2}) = -\tilde{R}_4^{(i)} + r_1^{(i)}, \tag{V.10}$$

$$t^{(i)} = -[P_3^{(i)}]^{-2}, \tag{V.11}$$

$$q^{(i)} = r_1^{(i)} + r_1^{(i-1)}, \tag{V.12}$$

$$r_1^{(i)} = P_1^{(i)}[P_3^{(i)}]^{-1}, \tag{V.13}$$

$$P_1^{(i)} = \cosh(\lambda^{(i)}h^{(i)}), \tag{V.14}$$

$$P_3^{(i)} = \lambda^{(i)}\sinh(\lambda^{(i)}h^{(i)}), \tag{V.15}$$

and

$$\lambda^{(i)^2} = D_{11}(r_i). \tag{V.16}$$

In the above equations, $h^{(i)}$ is the width of sector i and r_i is the center of sector i. Equations (V.8) and (V.10) need be evaluated only once per calculation, Eq. (V.9) is the recursive step and the other equations are vectorizable with vector lengths equal to the number of sectors, which is typically on the order of hundreds. Thus significant vector speedups are obtainable. It should be noted that as in the case of classical dynamics, the vectorized steps here do not map as well to the Cray architecture as operations such as matrix multiplication do, so that extremely high execution rates are not possible.

For calculations with large N, it will be more efficient to vectorize the operations required for the individual integration steps. The majority of the time will be spent on standard matrix manipulations on matrices of order N by N. For large N, these operations will dominate the calculation because they scale as N^3. Since (depending on the algorithm) various numbers of matrix multiplications, matrix inversions, linear equation solutions, and matrix diagonalizations will be required per integration step, the choice of a particular algorithm will be based on the relative work of the various operations, that is, the coefficient multiplying N^3, and also on the execution rate of the operations, which introduces another coefficient multiplying the operation count. We now consider these last points in detail.

In Table 12, we quote estimates of the asymptotic execution rates and matrix order required to achieve half the asymptotic rate for several matrix operations. The operation count for large enough N will scale as $C_x N^3$, with C_x a coefficient depending on the operation. For matrix multiplication we take $C_{MXM} = 2$, for general linear equation solution we use $C_{GLS} = 8/3$ ($2N^3/3$ for LU decomposition, and $2N^2$ per right hand side for forward elimination and back substitution), for symmetric linear equations we use half this value: $C_{SLS} = 4/3$. Finally for real symmetric matrix diagonalization, we estimate $C_{RS} = 10$. These operation counts are based on the discussions of Ref. 127. The data in the table were determined by averaging the results obtained by fitting the CPU times and MFLOPS rates with matrices of order 63, 127, 255 and 511 to $N^2(t_s + Nt_\infty)$ or $10^{-6}C_x/(t_\infty + t_s/N)$ by least squares. These particular orders where chosen to be close to multiples of 64 to make most efficient use of a Cray computer's 64-element vector registers while at the same time avoiding multiples of 2, which can cause catastrophic bank conflicts as discussed in Section III above. For example, the execution rate for 512 is a factor of 20 less than for 511 on the CRAY–2 when using the general linear equation routine LUSOLV. We should also point out that by fitting only these above array dimensions we obtain different $n_{1/2}$ values from those in section III above. In particular, the fit includes only array dimensions considerably larger than the true $n_{1/2}$ values for matrix multiplication. The fitted $n_{1/2}$results for this case are too small and should not be taken literally.

For matrix multiplication, we give results for the Cray routine MXM. For diagonalization, we only quote results for the EISPACK [57] routine RS, which we have found most convenient and reliable for our scattering calculations, although other routines may be more efficient [128]. In the following comparison we will emphasize asymptotic execution rates (r_∞), but for finite matrices the $n_{1/2}$ values are also important parameters.

For linear equations, we give results for four different routines. The first, called LUSOLV, is a FORTRAN code utilizing loops unrolled to a depth of 16 [14,129,130]. The next is the Cray library routine MINV [22]. The last two are LINPACK [131] routines, one for a general matrix and one for symmetric matrices. Comparing the four methods for linear equation solution, we see that it is never advantageous from a CPU time point of view to use the specialized symmetric matrix routine, for its execution rate is always much more than a factor of two less than the other routines. While it uses less memory, since only a triangle of the matrix needs be stored, the execution rates are so poor that this is unlikely to be of sufficient motivation to use it. On the X–MP/48, the execution rates of the general LINPACK routine and the library routine MINV are similar, but the LUSOLV program is about a factor of two more efficient then those routines. On the Y–MP the situation is similar, with the LUSOLV program being the fastest, followed by the general LINPACK routine, and then MINV, with relative ratios 1:1.2:2.5. On either CRAY–2 machine the situation changes, and the Cray library routine MINV is almost a factor of two faster then LUSOLV, and more than a factor of 2.5 times faster then the general LINPACK routine. Thus the most efficient routine to use on the X–MP or the Y–MP is the FORTRAN program LUSOLV, whereas on the CRAY–2 the most efficient routine is the CRI's MINV.

The relative efficiencies of the various operations can be estimated by comparing the r_∞ values (see also Section III). Since matrix multiplication is usually the most efficient operation, it is convenient to introduce the ratios $\mathcal{R}_{RS}$ and $\mathcal{R}_{LE}$ which measure the asymptotic rates compared to matrix multiplication for diagonalization and the best method for linear equation solution. On the X–MP and Y–MP we have $\mathcal{R}_{RS} = 1.2$ and $\mathcal{R}_{LE} = 1$, while on the CRAY–2 these ratios are 2.6 and 0.91-1.0, depending on the memory speed. On the X–MP and Y–MP, therefore, these various matrix operations are approximately equally efficient, while on the CRAY–2 the diagonalization code is much less efficient then the others. However on all Cray machines, the $n_{1/2}$ values are much shorter for matrix multiplication than for the other operations, so for small to moderate matrices, the diagonalization and linear equation solution operations will be relatively less efficient then the above discussion indicates. We should again point out that these timings (especially CRAY–2 values) are subject to about 10% fluctuation, depending on system loads.

Based on the timings reported above, it seems desirable to maximize the number of matrix multiplications and minimize the other operations in the overall scheme. One algorithm which does this, at least in principle, is the DeVogelaere method [132], which only requires matrix multiplications. However in practice, two aspects diminish the attractiveness of this method. First of all, the r step size is limited by the oscillation of the wave functions, that is, a certain number of integration steps will be required per De Broglie wavelength. Thus for long-range potentials or high energy collisions, an extremely large number of integration steps will be required. In contrast, a number of algorithms exist where the integration step size is controlled by the change in the interaction potential and thus large steps can be taken where the potential is slowly varying. The step sizes required for accurate results for these methods are only weakly dependent on total energy also.

The second difficulty with the DeVogelaere method is one common to all initial value methods that explicitly determine the radial functions $\tilde{\mathbf{f}}$. At small r, all channels will be classically inaccessible, and thus the functions will be growing exponentially. If the classically forbidden region is too large, then the component of the functions which has the largest exponential growth will become sufficiently larger than all other components that (to working precision) the linear independence of the initial conditions will be lost. The determination of the scattering matrix will then not be possible [132]. This can be controlled to a certain extent by periodically reorthogonalizing the columns of $\tilde{\mathbf{f}}$ [133], but this stabilization procedure introduces extra operations and a small integration step may be required to limit the exponential growth per step. In practice, for molecular collisions where it is necessary to include channels which are classically inaccessible for all r, this difficulty can be a severe problem.

An alternative solution is to devise an algorithm which does not explicitly determine the $\tilde{\mathbf{f}}$ but rather some other quantity which will be inherently stable. For example, the logarithmic derivative matrix $\tilde{\mathbf{f}}^{-1} d\tilde{\mathbf{f}}/dr$ will not suffer from stability problems.

Because of these numerical difficulties, we have found the R–matrix propagation algorithm to be very attractive [124-126]. In this method, the integration step size is usually limited by the r derivative of the coupling matrix $\mathbf{D}$, so large step sizes are possible in the asymptotic region where the interaction potential is slowly varying, which is particularly advantageous for long-range potentials. In addition, the negative of the inverse of the logarithmic derivative matrix is propagated, which avoids all stability problems. A final advantage is that a large fraction of the work required is independent of total energy E so that effort can be saved by performing calculations for several energies. A disadvantage is that operations other than matrix multiplications are required.

The operations involved per integration step for this algorithm are as follows. The energy independent steps are to construct the coupling matrix $\mathbf{D}$ (which is real and symmetric) in sector i, to diagonalize it to determine the local adiabatic eigenvectors $\mathbf{T}^i$ and eigenvalues $[\lambda^{(i)}]^2$, and then to form the overlap matrix

$$\mathcal{T}(i-1,i) = [\mathbf{T}^{(i-1)}]^t \mathbf{T}^{(i)}. \tag{V.17}$$

For calculations at subsequent energies, it is only necessary to save the local adiabatic eigenvalues and overlap matrices. Thus for large N, the relative time for the energy independent step will be $N^3(2 + 10\mathcal{R}_{RS})$. Since the total energy enters in Eq. (V.4) only as a multiple of the unit matrix, the local adiabatic eigenvectors are independent of E and the eigenvalues are shifted by a constant amount if E changes. Then at each energy it is necessary to form

$$\mathbf{R}_4^{(i)} = \mathbf{r}_4^{(i)} - [\mathbf{P}_3^{(i)}]^{-1}\{\mathbf{R}_4^{(i-1)}\mathcal{T}(i-1,i) + \mathcal{T}(i-1,i)\mathbf{r}_4^{(i)}\}^{-1}\mathcal{T}(i-1,i)[\mathbf{P}_3^{(i)}]^{-1}, \tag{V.18}$$

where

$$\mathbf{R}_4^{(i)} = -\mathbf{f}[d\mathbf{f}/dr]^{-1}|_{r=r_i+h_i/2}, \tag{V.19}$$

$$\mathbf{r}_4^{(i)} = [\mathbf{P}_3^{(i)}]^{-1}\mathbf{P}_1^{(i)}, \tag{V.20}$$

$$[\mathbf{P}_1^{(i)}]_{nn'} = \delta_{nn'}\cosh(\lambda_n^{(i)}h_i), \tag{V.21}$$

and

$$[\mathbf{P}_3^{(i)}]_{nn'} = \delta_{nn'}\lambda_n^{(i)}\sinh(\lambda_n^{(i)}h_i). \tag{V.22}$$

Since $\mathbf{P}_1^{(i)}$, $\mathbf{P}_3^{(i)}$ and $\mathbf{r}_4^{(i)}$ are diagonal, the time-consuming operations required for (V.18) are one matrix multiplication and one linear equation solution, and so we see that the relative time for the energy dependent step is $N^3(2 + 8/3\mathcal{R}_{LE})$. If we consider large scale calculations on the CRAY–2, where we take $\mathcal{R}_{RS}$ as 2.6 and $\mathcal{R}_{LU}$ as one, then we see that the ratio of the energy independent time to the energy dependent time is six, thus it is advantageous to perform calculations at many energies — performing calculations at seven energies would require only about twice as much CPU time as one energy. (Note that an earlier version of our code [130, 134] used a different sequence of matrix operations which was less efficient both in time and memory usage.) This assumes that the time for the evaluation of the matrix elements of the interaction potential required for (V.4) is negligible. This is not always the case, even though the work to construct this matrix should scale as N^2 for large enough N. If the interaction potential time is not negligible, then the ratio of the energy dependent times to the energy independent times will be even greater. Strategies for optimizing the interaction potential matrix evaluation will be discussed below.

The storage requirements for the R–matrix propagation algorithm are relatively small, considering the amount of memory available on the CRAY–2. The matrices which need to be stored are of order $N \times N$, and the number required is not large. For a single energy calculation, our present code requires seven matrices of this size, excluding the data required to form the interaction potential matrix. Usually we keep these matrices in memory, but some calculations carried out on a CRAY–1 used a version of the code which kept only two matrices in core and the remainder on disk. For the values of N we used in those calculations (440 and 530), the I/O requirements of this strategy did not significantly degrade the performance of the method. This would be especially true on a machine with a large SSD. Other quantities which need to be stored are those required to evaluate the potential coupling matrix. The requirements here may or may not be considerable, depending on the potential used. In our large scale HF + HF calculations, using a complicated potential, our code required approximately an additional $40N^2$ words of storage for angular integrals and pointers (see below for a description of the construction of the potential matrix). Fortunately, this data is accessed sequentially, thus comparatively little is lost by storing it on secondary storage like disk or an SSD.

If more than one energy is to be used, there are two storage choices. The first choice is to finish the calculation at the first energy before performing any part of the calculation for other energies. This requires that an additional N_{step} matrices be saved (the sector overlap matrices), where N_{step} is the number of integration sectors used. The second choice is to perform the calculations for all energies at sector i before going on to the next sector. This requires $8 + N_E$ matrices, where N_E is the number of energies used [134]. Thus since N_{step} is usually on the order of hundreds and N_E on the order of tens, it is usually more efficient to make the second choice.

We now turn to the question of evaluating the interaction potential matrix. It is usually most efficient to transform to the body-fixed frame of reference to evaluate the interaction potential matrix. In this frame the kinetic energy part of $\mathbf{D}$ is no longer diagonal, but the coupling is very simple [135]. It is necessary to back-transform to the laboratory frame only when the asymptotic analysis to determine the scattering matrix is carried out. For A + BC collisions we need to compute the body-frame matrix elements

$$< vj\Omega JMP|V^{int}|v'j'\Omega JMP >=2\pi \int dR \int d\cos\chi \; Y^*_{j\Omega}(\chi,0)\Theta^*_{vj}(R)V^{int}(r,R,\chi) \\ \times Y_{j'\Omega}(\chi,0)\Theta_{v'j'}(R) \tag{V.23}$$

where v and v' are vibrational quantum numbers, j and j' the diatomic rotational angular momentum quantum numbers, J the total angular momentum, M is the projection of J on the laboratory-frame z axis, Ω the projection of j, j' and J on the body-frame z axis, P is the parity, R is the diatomic bond length, r the distance between A and the center of mass of BC, χ the angle between the vectors from A to the center of mass of BC and C to the center of mass of BC, $Y_{j\Omega}$ is a spherical harmonic, and Θ_{vj} is a vibrational function. To arrive at (V.23) we have averaged over the Euler angles rotating an arbitrary laboratory-fixed coordinate system to the body fixed coordinate system: this produces a Kronecker delta for Ω, J, M, and P, and the factor 2π. The traditional way to proceed is to expand the $\cos\chi$ dependence of the interaction potential in terms of Legendre polynomials and then perform the angular integrals analytically. The R integral is then performed most efficiently using a optimized quadrature [136] so that the matrix elements become

$$< vj\Omega JMP|V^{int}|v'j'\Omega JMP >= \sum_{i,\lambda} w_{vjv'j',i} \; v_\lambda(r,R_i) f^{\Omega}_{jj'\lambda}, \tag{V.24}$$

where $w_{vjv'j',i}$ is a vibrational quadrature weight, v_λ is a potential expansion coefficient, and $f^{\Omega}_{jj'}$ is an angular integral (which is independent of M, J, and P). In general, it will also be necessary to generate the potential expansion coefficients, and this is most straightforwardly done by projection:

$$v_\lambda = \frac{2\lambda+1}{2} \int d\cos\chi \; P_\lambda(\cos\chi)V^{int}(r,R,\chi). \tag{V.25}$$

The quadrature approximation to (V.25) is most efficiently evaluated as a matrix-vector product. Usually if the potential is given explicitly in terms of a Legendre expansion there are only a few terms in the λ sum, however, if it is necessary to converge the angular expansion of a more general potential several tens of terms are typically required. The difficulty of efficiently evaluating (V.24) arises for two reasons. First, the angular integrals are zero unless the triangle rule for j, j', and λ is satisfied, thus many integrals are zero and it is advantageous to exploit that. Second, the way in which two pieces of information depending only on a subset of the quantum numbers are combined together complicates efficient evaluation.

We now consider three schemes to evaluate (V.23). In scheme A we evaluate (V.25) by matrix-vector product and then form

$$\tilde{v}_{vjv'j',\lambda}(r) = \sum_i w_{vjv'j',i}\, v_\lambda(r, R_i). \tag{V.26}$$

This is performed as a matrix multiplication with $vjv'j'$ labeling rows and λ columns of the result. The inner index i is the number of points in the vibrational quadratures and usually is on the order of ten. Finally a sparse SAXPY is performed for each value of λ to generate the matrix element. In scheme B the procedures of A are modified by forming the intermediate product

$$A^{\Omega}_{jj'}(r, R_i) = \sum_\lambda v_\lambda(r, R_i) f^{\Omega}_{jj'\lambda}, \tag{V.27}$$

which is a sparse SAXPY followed by

$$< vj\Omega JMP|V^{int}|v'j'\Omega JMP >= \sum_i w_{vjv'j',i}\; A^{\Omega}_{jj'}(r, R_i), \tag{V.28}$$

which is a vector plus vector times vector using scattered data. Finally in scheme C we dispense with the expansion of the potential and directly [†] form the $A^{\Omega}_{jj'}(r, R_i)$ via

$$A^{\Omega}_{jj'} = \sum_n C^{\Omega}_{jn} \tilde{C}^{\Omega}_{nj'}(r, R_i), \tag{V.29}$$

where

$$\tilde{C}^{\Omega}_{nj} = C^{\Omega}_{jn} V^{int}(r, R_i, \cos\chi_n), \tag{V.30}$$

and

$$C^{\Omega}_{jn} = \sqrt{2\pi\tilde{w}_n} Y_{j\Omega}(\chi_n, 0), \tag{V.31}$$

$\tilde{w}_n$ a Gauss-Legendre quadrature weight. Equation (V.29) is evaluated as a matrix multiplication. The number of points in the angular quadrature is on the order of ten.

[†] We are grateful to J. N. Murrell for suggesting this option.

The three schemes have various advantages and disadvantages. A drawback of schemes A and B is that is it necessary to form the angular integrals $f^{\Omega}_{jj'\lambda}$, while in scheme C it is necessary to form the C^{Ω}_{jn}, which is an easier task and can be done efficiently in vector mode. However, these steps are independent of r and R and so should not be of significant overall cost. In Scheme A the vibrational integral is computed very efficiently, but too much work is performed: many of the $\tilde{v}_{vjv'j',\lambda}$ will not be required since many of the $f^{\Omega}_{jj'\lambda}$ are zero. Schemes B and C perform no more work than is required for the vibrational integrals, but (due to the scattering of data because a given $vjv'j'$ pair contributes to more than one Ω) this work will not be performed as efficiently in scheme A.

Turning now to the angular part of the different methods, scheme A only performs the sparse SAXPYs once per λ, while scheme B performs them once per λ per vibrational quadrature point. Scheme C also needs to repeat the angular integrals for each vibrational quadrature point. However, if there were only one vibration quadrature point, scheme A would perform more work because the total number of angular integrals (zero and nonzero) per λ is proportional to the square of the number of vj states, whereas for schemes B and C the number is proportional to the square of the number of j states. Thus the relative efficiencies of schemes A and B will depend on particular circumstances, with a larger number of v states coupled with a smaller number of vibrational quadrature points favoring scheme B and a small number of v states using many vibrational quadrature points favoring scheme A. In our experience, usually the efficiency performing the vibrational integrals is the most important factor, thus scheme A is more efficient than scheme B. This is another example of how minimizing the operation count does not necessarily lead to the most efficient algorithm on vector computers.

In most cases, scheme B will be preferred over scheme C because the matrix multiplication to produce the angular integrals takes longer than the sparse SAXPYs. However, in special cases scheme C also has one potentially important advantage: it may be possible to greatly reduce the number of vj states used in the wave function expansion by using some sort of adiabatic rotational states rather than spherical harmonics [137]. That is, a few new functions $\tilde{Y}_{n\Omega}$, which are linear combinations of many of the $Y_{j\Omega}$, could be defined and used instead.

This may allow otherwise intractable problems to be solved. A single left transformation of the C^{Ω}_{jn} is all that is required for scheme C, while the other schemes require similarity transformations for each value of λ to produce new angular integrals which now in general are not sparse, so the sparse SAXPYs are replaced with full SAXPYs. This will increase both the memory and the operation count for these schemes. The transformation times will be important because the transformations depend on r and hence must be carried out many times.

Turning now to the case for AB + CD collisions, the matrix element to be found is [138] $< v_1 v_2 j_1 j_2 j_{12} \Omega JMP | V^{int} | v_1' v_2' j_1' j_2' j_{12}' \Omega JMP >$, where v_i is the vibrational quantum number of molecule i, j_i is the rotational quantum number of molecule i, j_{12} is the quantum number of the vector sum of j_1 and j_2, J is the total angular momentum quantum number, M is its projection on the laboratory frame z axis, Ω is the projection of j_{12}, j_{12}', and J on the body fixed z axis, and P is the parity. The procedure here is very similar to the A + BC case, except that the manipulations are more complicated. Expanding the angular dependence of the potential in terms of combinations of spherical harmonics now involves three angles and three indices, thus the number of terms required is approximately the cube of that required for the atom-diatom case [139]. For anisotropic potentials, close to a thousand terms can be required to converge the angular representation of the potential [140]. Thus although one may be tempted to ignore the presence of zero angular integrals in the A + BC case to simplify vectorization, for the AB + CD case the number of integrals is so large that it is imperative that the zero integrals be identified and not stored. The aggregate number of angular quadrature points can also be in the thousands. The situation for the vibrational integrals is not quite as bad, because there are only two vibrational coordinates and so the number of quadrature points will be only the square of the A + BC case, say, 50 to 100 if optimized quadratures [136] are used. The three different schemes described above for performing the integrals can be applied here with little modification with the exception of scheme C which for Ω greater than zero must be modified to include contributions from two numerical integrals, that is, two matrix multiplications are required, because of the more complicated coupling of the angular momentum vectors. Thus C^{Ω}_{jn} of (V.31) is replaced with the quantity $C^{\Omega x}_{j_1 j_2 j_{12} n}$ given by

$$C^{\Omega x}_{j_1 j_2 j_{12} n} = 2[\pi \tilde{w}_{n_1} \tilde{w}_{n_2} \bar{w}_{n_3}]^{\frac{1}{2}} \sum_{m_1 m_2} (j_1 m_1 j_2 m_2 | j_1 j_2 j_{12} \Omega) N_{j_1 |m_1|} (-1)^{\frac{m_1+|m_1|}{2}}$$
$$N_{j_2 |m_2|} (-1)^{\frac{m_2+|m_2|}{2}} P^{|m_1|}_{j_1}(\cos \chi_{n_1}) P^{|m_2|}_{j_2}(\cos \chi_{n_2}) x[\frac{\phi_{n_3}}{2}(m_2 - m_1)] \tag{V.32}$$

where x is either sin or cos, n is a composite index denoting the three quadrature indices n_1, n_2 and n_3, $\tilde{w}_n$ and $\bar{w}_n$ the quadrature weights for the different angular integrations, $(...|...)$ is a Clebsch-Gordan coefficient, N_{jm} the (positive) normalization factor for the spherical harmonic Y_{jm}, P^m_j is an associated Legendre function, χ_1 the center of mass BC to center of mass AB to B angle, χ_2 the inclination angle for the second molecule, and ϕ is the dihedral angle. If Ω is zero, then only the functions with x = cos are required. For Ω greater than zero, the two matrix multiplications are for cosine terms times cosine terms and sine terms times sine terms. The weighting of the various schemes changes somewhat for this case because of the relative complexity of the analytic calculation of the angular integrals required for schemes A and B. Although these integrals are still independent of r, their calculation can be time-consuming enough that they can consume a significant fraction of the computational resources required for a scattering calculation. The analytic formulas involve 3-j, 6-j and 9-j symbols which are generated mostly in scalar mode. In contrast, scheme C, which only requires the functions of (V.32), involves only a Clebsch-Gordan coefficient (which is a rephased and normalized 3-j symbol), thus the time to calculate the functions for scheme C will be much less than time to form the angular integrals required for the other two schemes. In addition, the efficient implementation of adiabatic rotational states possible with scheme C makes it rather attractive.

An alternative scheme to reduce the operations required to evaluate the **D** matrix is to employ some form of discrete variable representation [141]. Here a transformation is made so that the potential coupling is diagonal with elements equal to the potential at grid points. The kinetic energy is not diagonal in this basis. So far this method has been applied only to rotational-like degrees of freedom for three atom systems. Another way to reduce the effort to evaluate the potential matrix is to write the interaction potential in a special form whereby much of the angular and vibrational integrations need be done only once [122].

We also reiterate that the potential can be reused for calculations at more than one energy, and if the body-frame matrix elements are used as discussed above, they can be used for more than one value of the total angular momentum J. Finally it may be possible to achieve speedups by evaluating the potential matrix using a coarser grid then is used to integrate (V.3) and then interpolating the result. Thus if one is willing to perform extensive enough calculations (many energies, many J), the evaluation of the potential matrix will be an insignificant part of the overall calculation, and the overall execution rate is determined by the rate of the matrix manipulations in the algorithm used for solving (V.3).

We now turn to some new algorithms and methods which have been made viable by the availability of the large CRAY–2 memory. In contrast to the quantum mechanical methods discussed so far, which propagate a solution along r, and hence need only store the relevant data for one value of r, the methods we mention now are more global in nature and require data for all values of r. The first method we discuss is the finite difference boundary value method (FD-BVM) [142,143]. Here the close-coupling equations are solved not by specifying two sets of boundary conditions at small r, but rather by specifying a boundary condition at both small and large r. This can be done in many ways, and the easiest is to set up a grid of r consisting of the points r_1, r_2, ... $r_{N_{grid}}$ and then approximate the derivative operator in (V.3) by finite differences, for example

$$\frac{d^2}{dr^2}\tilde{\mathbf{f}}(r_i) = \sum_j C_{ij}\tilde{\mathbf{f}}(r_j). \tag{V.33}$$

This converts the differential equation into a set of linear equations of the form

$$\mathbf{\Lambda}\vec{\mathbf{f}} = \vec{\beta}, \tag{V.34}$$

with $\mathbf{\Lambda}$ a band matrix made up of the $\mathbf{D}(r_i)$ and C_{ij}, $\vec{\mathbf{f}}$ the vector made up of a particular column of $\tilde{\mathbf{f}}$, say column n_o, at each of the N_{grid} distances, and $\vec{\beta}$ consists of values of column n_o of $\tilde{\mathbf{f}}$ for distances less than r_1 and greater than $r_{N_{grid}}$. For r less than r_1 we set $\tilde{\mathbf{f}}$ to zero. For r greater than $r_{N_{grid}}$, the boundary conditions are more complex, because they require knowledge of the scattering matrix, which is not available at this step of the calculation. Thus we are restricted to finite difference approximations which require knowledge of the solution only at one distance beyond $r_{N_{grid}}$. Then an arbitrary normalization can be introduced into (V.7) and we can take

$$\tilde{f}_{nn_o}(r_{N_{grid}+1}) = \delta_{nn_o}. \tag{V.35}$$

The bandwidth of $\mathbf{\Lambda}$ depends on the number of points in the finite difference approximation, and in our calculations we have found it efficient to use a 9-point approximation everywhere except for the large r end of the grid where the number of points is reduced to seven and then eventually down to three in order that only one point beyond the end of the grid is required for the boundary conditions. Of course this necessitates using a smaller step size for the last grid points in order that the error of the solution not be dominated by the error in the final 3-point approximation [144]. With this scheme the half-bandwidth of $\mathbf{\Lambda}$ is equal to $4N$, where N is the number of channels included in the close-coupling equations.

There are several advantages of this method which offset to a certain degree the extra storage resources required by it. Perhaps the most important feature is the high quality of radial functions produced. In contrast to initial value methods which directly determine the radial functions, the FDBVM has no difficulties with instability due to classically forbidden regions. Thus if rather than just requiring the scattering matrix, one is interested in using the radial functions for other purposes, such as in integrals, then the FDBVM may be the method of choice. An important factor in making this worthwhile is the fact that there is no restriction on the grid points, that is, they can be unevenly spaced. We have used this feature to advantage by including Gaussian quadrature points in the finite difference grid and then used the resulting radial functions in numerical integrals using efficient Gaussian quadrature rules [144]. Another feature of the method is that it is very easy to solve inhomogeneous versions of (V.3), because the inhomogeneity will simply appear in $\vec{\beta}$. In addition, because the work to solve (V.34) is primarily made up of forming the LU decomposition of $\mathbf{\Lambda}$, then it is possible to solve for several inhomogeneities for little extra cost once the homogeneous problem has been solved [145]. The final advantage is that (V.34) can be solved by reasonably efficient black box programs, so that good execution rates can be obtained even for relatively small N, in contrast to the step by step methods discussed above. Another aspect of this is that most of the time will be spent solving one set of linear equations, so the user need not be concerned with optimizing large amounts of code, but simply calling the library routine.

We next turn to methods which are not based on (V.3) but rather on equivalent integral equations. The number of methods here is very large [146-148], but the basic ideas are similar. Rather than numerically obtaining the radial functions by integrating a differential equation, the radial functions (or related quantities) are expanded in terms of known basis functions, and then the unknown coefficients are determined from the solution of a single set of linear equations, or the scattering matrix and related quantities are determined directly from matrix elements over the basis functions using linear algebra. Thus the work in forming the scattering matrix can be broken down into two parts, the calculation of the matrix elements and the solution of the linear equations. If there are an average of M basis functions per each of the N channels, then for large enough $M \cdot N$, the work for the matrix element calculation will scale as $(M \cdot N)^2$ while the work for the linear equations step will scale as $(M \cdot N)^3$. Thus rather then performing many matrix operations which scale as N^3, there is one large operation scaling as $(M \cdot N)^3$, so the size of M^3 compared to the number of integration steps will be important in determining the relative efficiency of these methods compared to the methods based on differential equations. However, these basis function methods have several advantages which can counterbalance inefficiencies and make them the methods of choice. A particular example is in the area of rearrangement collisions. The quantum mechanical equations of motion based on the differential equation approach lead in this case to integro-differential equations [149], which are very difficult to solve. Although it is possible by a suitable choice of coordinates to turn the rearrangement equations of motion into differential equations [150], new complexities arise [151,152]. In contrast, the application of basis function methods leads only to exchange integrals which involve all known functions, so the methodology is virtually unchanged [144]. Another potential advantage is that the linear equations can be solved iteratively, so the work per initial state will scale as $m(M \cdot N)^2$, where m is the number of iterations required [153-155]. This is an advantage when one is interested in transitions out of only a few initial states, so only a few columns of the scattering matrix are needed; however, all of the methods which we have discussed so far use arbitrary boundary conditions and hence require the generation of N linearly independent solutions in order to determine any column of the scattering matrix. Another potential advantage is based on the way scattering calculations are usually performed. In order to assess the convergence of the calculations, it is necessary to perform a sequence of calculations which differ in the number of basis functions used, and thus many problems which are very similar are solved. What we wish to do is use some of the information from smaller, preliminary calculations, to make the larger calculations less expensive. One way to do this is to form linear combinations of the M basis functions based upon smaller calculations and use a smaller number of these contracted functions in the linear equation step [156]. Even if the number of basis functions per channel can be reduced only slightly, the cubic operation count will magnify this reduction and make it a worthwhile step. The parallels here with improving the basis functions in electronic structure calculations (such as the use of atomic natural orbitals) are striking.

These basis function methods are still quite new for applications to heavy particle collisions, and are undergoing rapid development. One important area which needs further work is the efficient evaluation of the matrix elements. While the scaling arguments presented above show that this will be a negligible step for large enough calculations, experience to date is that the coefficient multiplying the $(M \cdot N)^2$ factor is large enough so that it can dominate the calculation. However, the best implementation here is probably yet to appear.

All of this discussion relies on the assumption that the resources for the calculations exist, and this means primarily memory. If a calculation based on the differential equation approach, which only requires the storage of matrices of order N^2, taxes the memory of a machine, clearly the basis function approach is not practical. For this reason, almost all converged calculations of three-dimensional heavy particle collisions have been performed on the CRAY–2.

We now turn to the final method to be considered. Here we move from the time-independent Schrödinger equation to the time-dependent form. This introduces many new complexities into the calculations, but at the same time several advantages appear. The basic idea is that the time-dependent Schrödinger equation is first order in time, so the initial conditions determine the solution for all other times [157]. This in turn requires that the calculations are for a particular initial quantum state, in contrast to most of the methods discussed above. The initial condition consists of a mixed basis function/grid representation of the wave function, and the initial translational energy range is determined via the uncertainty principle from the spatial localization of the wave function. The solution is propagated forward in time until after the collision is over and the results are analyzed to determine the probabilities for the various final states. These calculations are both computationally intensive and consume considerable storage resources, so most large-scale calculations have been carried out on the CRAY–2. There are three main advantages of the method. First, as mentioned above, the method only considers a single initial state. Thus the scaling of the operations of the method will be less than the third power, unlike the previous methods discussed, and so for large enough calculations this method will be more efficient. Second, since the initial conditions consist of a spread of initial energies, it is possible to obtain results for many different energies from a single time propagation run. Finally, since the time-dependent equation is solved and one is explicitly dealing with wave packets, the physical interpretation of the results can bring more insight than for the time-independent methods.

C. The influence of supercomputers on dynamics calculations.

Relatively few classical dynamics studies utilizing vectorized codes running on Cray computers have been reported in the literature, mainly because the calculations can be carried out on slower machines. Indeed, the time per trajectory is small enough that even personal computers have been used. What contributes to the overall time is the number of trajectories required, and the advantage of a vectorized code is that results can be obtained in much less real time (perhaps days rather than months). This offers the opportunity for much better feedback to others whose work depends on the outcome of the dynamics, just as discussed in section IV for the electronic structure case.

In the area of quantum dynamics, the advent of supercomputers has opened entire new horizons. The available processing power is beginning to make the study of non-reactive collisions in systems of more than three atoms feasible; this will allow the investigation of phenomena like vibrational-to-vibrational energy transfer, a process which cannot occur for less than four atoms. The large memory of the CRAY–2 has been the main impetus behind a renaissance in quantum reactive scattering, for the application of the basis function methods that have proved so fruitful would not be practical on smaller machines.

VI. Conclusions and future directions

We have reviewed the performance of essentially all the currently available Cray computers on typical tasks in quantum chemistry and dynamics. Our discussions show that it is possible to achieve a large fraction of the possible machine performance in all phases of these calculations, and we have also considered the impact these developments have had on research in quantum chemistry and dynamics. In view of the fact that one consequence of an increase in available computing power is to whet researchers' appetites for even more computing power, we shall discuss briefly here some future directions.

The most important development in the next few years is likely to be a much greater use of multitasking on Cray computers. As we have discussed here and elsewhere [28], multitasking will probably be a necessity to ensure that all CPUs are kept busy in multiple CPU systems, and it should also be used when a single job needs access to a large fraction of system resources. The arrival of microtasked library subroutines should encourage more multitasking, as will the incorporation of automatic multitasked code generation into the CFT77 compiler, but some burden will fall on the programmer if coarse-grained parallelism is to be exploited. There has been little work reported investigating the use of multitasking on normal production machines: our own experiences suggest that some work remains to be done at the operating system level in implementing multitasking. Given that there is the potential on the Y–MP to achieve close to 2.5 GFLOPS using all eight CPUs, as discussed in section IIID, the rewards from multitasking can be considerable, and we can expect to see much work in this area in the near future.

Another important development will be the arrival of new supercomputer models. The CRAY–3, featuring 16 CPUs and a clock period of 2 ns and using gallium arsenide technology, has been described in some detail [158]. The theoretical maximum performance would be 1 GFLOPS per CPU, or 16 GFLOPS using all CPUs together. This machine is an obvious development from the CRAY–2, and much of the experience with the latter should carry over to the CRAY–3. There is already discussion of the CRAY–4 [158], which is planned to out-perform the original CRAY–1 by a factor of 1 000 — three orders of magnitude in some twenty years. These new machines will have memories of 1 GW and larger, which will stimulate not only the wider use of some of the large-memory-based algorithms we have described in sections IV and V, but also the development of new schemes that can exploit even larger memories. This will undoubtedly lead to novel and exotic approaches to many problems, but the more commonplace methods described earlier will also see substantial performance benefits from the newer machines. It should be noted, however, that the size of electronic structure problems that can be tackled using conventional methods is commonly limited by external storage capacity even on the current range of Cray computers, and this limitation will only become more acute as faster machines appear. While methods designed to circumvent this limitation will undoubtedly be developed, it is very likely that providing adequate external storage will be a major problem for supercomputer vendors in the next decade.

Having reviewed developments in hardware and software for computational chemistry on Cray computers, a final conclusion can be drawn. The power of Cray supercomputers, with their particular suitability for computational chemistry, has stimulated many important and fundamental developments in the field, and as the next generations of Cray computers appear we can be confident that this trend will continue. There is therefore every reason for optimism about what the rather natural pairing of computational chemistry and Cray supercomputers will bring forth.

Acknowledgments

We would like to acknowledge helpful discussions with many colleagues: J. Almlöf, J. Avila, D. H. Bailey, J. Barton, L. A. Barnes, T. U. Helgaker, A. Komornicki, S. R. Langhoff, T. J. Lee, H. Partridge, J. E. Rice, and P. Siegbahn. E. Wimmer and E. A. Kroll of CRI made it possible for us to obtain information about the CRAY–3 and CRAY–4. We would also like to acknowledge the assistance provided by ZeroOne systems, the Advanced Computational Facility, and the Numerical Aerodynamic Simulation Facility at NASA-Ames. We are especially grateful to the NAS Facility for early access to the CRAY Y–MP. PRT was supported by NASA grant NCC 2-371.

REFERENCES

1. *CRAY X-MP and CRAY Y-MP Multitasking Programmer's Manual (publication SR-0222),* Cray Research Inc., Mendota Heights (1988).

2. *CRAY-2 Multitasking Programmer's Manual (publication SN-2026),* Cray Research Inc., Mendota Heights (1988).

3. *Advances in Chemical Physics, Vols 67 and 69,* John Wiley, New York (1988).

4. R. B. Bernstein, ed, *Atom-Molecule Collision Theory,* Plenum Press, New York (1979).

5. M. Baer, ed, *Theory of Chemical Reaction Dynamics, Vols 1-5,* CRC Press, Boca Raton (1985).

6. R. D. Levine and R. B. Bernstein, *Molecular Reaction Dynamics and Chemical Reactivity,* Oxford University Press, Oxford (1987).

7. *COS Reference Manual (publication SR-0011),* Cray Research Inc., Mendota Heights (1987).

8. *UNICOS User Commands Reference Manual (publication SR-2011),* Cray Research Inc., Mendota Heights (1988).

9. C. W. Bauschlicher, A. Komornicki, H. Partridge, and P. .R. Taylor, unpublished work.

10. M. Metcalf and J. Reid, *FORTRAN 8X explained,* Oxford University Press, Oxford (1987).

11. *FORTRAN (CFT) Reference Manual (publication SR-0009),* Cray Research Inc., Mendota Heights (1987).

12. *CRAY-2 FORTRAN (CFT2) Reference Manual (publication SR-2007),*

Cray Research Inc., Mendota Heights (1987).

13. *CFT77 Reference Manual (publication SR-0018),* Cray Research Inc., Mendota Heights (1987).

14. J. J. Dongarra and S. C. Eisenstat, Squeezing the most out of an algorithm in Cray FORTRAN, *ACM Trans. Math. Software* **10**, 219-230 (1984).

15. *FX/FORTRAN Programmer's Handbook,* Alliant Computer Systems Corp., Littleton (1987).

16. *FX/SERIES Scientific Library,* Alliant Computer Systems Corp., Littleton (1987).

17. *Programmer's Library Reference Manual (publication SR-0113),* Cray Research Inc., Mendota Heights (1987).

18. V. R. Saunders and M. F. Guest, Applications of the CRAY–1 for quantum chemical calculations, *Comp. Phys. Comm.* **26**, 389-295 (1982).

19. C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh, Basic Linear Algebra Subprograms for FORTRAN usage, *ACM Trans. Math. Software* **5**, 308-323 (1979).

20. J. J. Dongarra, J. Du Croz, I. Duff, and S. Hammarling, *Preliminary Proposal for a Set of Level 3 BLAS,* Argonne National Laboratory (1987).

21. R. W. Hockney and C. R. Jesshope, *Parallel Computers — architecture, algorithms and programming,* Adam Hilger, Bristol (1980).

22. *Library Reference Manual (publication SR-0014),* Cray Research Inc., Mendota Heights (1986).

23. D. A. Calahan, P. L. Berry, G. C. Carpenter, K. B. Elliott, U. M. Fayyad, and C. M. Hsiao, *MICHPAK: A scientific library for the CRAY-2,* University of Michigan Report SARL 8 (1985).

24. V. Strassen, Gaussian elimination is not optimal, *Numer. Math.* **13**, 354-356 (1969).

25. V. Pan, New fast algorithm for matrix operations, *SIAM J. Comput.* **9**, 321-342 (1980).

26. D. Coppersmith and S. Winograd, On the asymptotic complexity of matrix multiplication, *SIAM J. Comput.* **11**, 472-492 (1982).

27. D. H. Bailey, Extra high speed matrix multiplication on the CRAY–2, *SIAM J. Sci. Stat. Comput.* **9**, 603-607 (1988).

28. P. R. Taylor and C. W. Bauschlicher, Strategies for obtaining the maximum performance from current supercomputers, *Theoret. Chim. Acta* **71**, 105-115 (1987).

29. S. R. Bourne, *The UNIX system,* Addison-Wesley, Reading (1983).

30. C. W. Bauschlicher, Considerations in vectorizing the CI procedure, in: *Advanced Theories and Computational Approaches to the Electronic Structure of Molecules* (C. E. Dykstra, ed), 13-18, Reidel, Dordrecht, (1984).

31. MOLECULE is a vectorized Gaussian integral program written by J. Almlöf.

32. J. Almlöf and P. R. Taylor, unpublished work.

33. SWEDEN is a vectorized SCF-MCSCF, direct CI, conventional CI-CPF-MCPF program, written by P.E.M. Siegbahn, C. W. Bauschlicher, B. Roos, P. R. Taylor, A. Heiberg, J. Almlöf, S. R. Langhoff, and D. P. Chong.

34. D. J. Fox, Y. Osamura, M. R. Hoffmann, J. F. Gaw, G. Fitzgerald, Y. Yamaguchi, and H. F. Schaefer, Analytic energy second derivatives for general correlated wave functions, including a solution of the first-order coupled-perturbed configuration-interaction equations, *Chem. Phys. Lett.* **102**, 17-22 (1983).

35. J. Almlöf and P. R. Taylor, Molecular properties from perturbation theory: a unified treatment of energy derivatives, *Int. J. Quantum Chem.* **27**, 743-768 (1985).

36. T. U. Helgaker and P. Jørgensen, Analytical calculation of geometrical derivatives in molecular electronic structure theory, *Adv. Quantum Chem.* **19**, 183-245 (1988).

37. *Optimization Guide (publication SN-0220),* Cray Research Inc., Mendota Heights (1983).

38. V. R. Saunders, Molecular integrals for Gaussian functions, in: *Methods in Computational Molecular Physics* (G. H. F. Diercksen and S. Wilson, ed), 1-36, Reidel, Dordrecht, (1983).

39. D. Hegarty and G. van der Velde, Integral evaluation algorithms and their implementation, *Int. J. Quantum Chem.* **23**, 1135-1153 (1983).

40. D. Hegarty, Evaluation and processing of integrals, in: *Advanced Theories and Computational Approaches to the Electronic Structure of Molecules* (C. E. Dykstra, ed), 39-66, Reidel, Dordrecht, (1984).

41. R. C. Raffenetti, General contraction of Gaussian atomic orbitals: core, valence, polarization, and diffuse basis sets; molecular integral evaluation, *J. Chem. Phys.* **58**, 4452-4458 (1973).

42. T. H. Dunning and P. J. Hay, Gaussian basis sets for molecular calculations, in: *Modern Theoretical Chemistry, Vol. 3, Methods of electronic structure theory* (H. F. Schaefer, ed), 1-27, Plenum, New York, (1977).

43. H. Taketa, S. Huzinaga and K. O-ohata, Gaussian-expansion methods for molecular integrals, *J. Phys. Soc. Japan* **21**, 2313-2324 (1966).

44. S. F. Boys, Electronic wave functions I. A general method of calculation for the stationary states of any molecular system, *Proc. Roy. Soc.* **A200**, 542-554 (1950).

45. L. E. McMurchie and E. R. Davidson, One- and two-electron integrals over Cartesian Gaussian functions, *J. Comput. Phys.* **26**, 218-231 (1978).

46. S. Obara and A. Saika, Efficient recursive computation of molecular integrals over Cartesian Gaussian functions, *J. Chem. Phys* **84**, 3963-3974 (1986).

47. J. Almlöf and P. R. Taylor, Computational aspects of direct SCF and MC-SCF methods, in: *Advanced Theories and Computational Approaches to the Electronic Structure of Molecules* (C. E. Dykstra, ed), 107-125, Reidel, Dordrecht, (1984).

48. J. Almlöf, University of Stockholm Institute of Physics Report 74-29 (1974).

49. R. M. Pitzer, Contribution of atomic orbital integrals to symmetry orbital integrals, *J. Chem. Phys.* **58**, 3111-3112 (1973).

50. E. R. Davidson, Use of double cosets in constructing integrals over symmetry orbitals, *J. Chem. Phys.* **62**, 400-403 (1975).

51. J. Almlöf and P. R. Taylor, General contraction of Gaussian basis sets. I. Atomic natural orbitals for first- and second-row atoms, *J. Chem. Phys.* **86**, 4070-4077 (1987).

52. M. Yoshimine, Construction of the Hamiltonian matrix in large configuration interaction calculations, *J. Comput. Phys.* **11**, 449-454 (1973).

53. C. C. J. Roothaan and P. S. Bagus, Atomic self-consistent field calculations by the expansion method, *Meth. Comp. Phys.* **2**, 47-94 (1963).

54. C. W. Bauschlicher and H. Partridge, Vectorizing the sparse matrix vector product on the CRAY X–MP, CRAY–2, and CYBER 205, *J. Comput. Chem.* **8**, 636-644 (1987).

55. F. W. Bobrowicz and W. A. Goddard, The self-consistent field equations for generalized valence bond and open-shell Hartree-Fock wave functions, in: *Modern Theoretical Chemistry, Vol. 3 Methods of electronic structure theory* (H. F. Schaefer, ed), 79-127, Plenum Press, New York, (1977).

56. P. S. Bagus, Energy expressions for open shell configurations of linear molecules for use in SCF calculations, IBM Report RJ 1077 (1972).

57. B. T. Smith, J. M. Boyle, J. J. Dongarra, B. S. Garbow, Y. Ikebe, V. C. Klema and C. B. Moler, *Matrix Eigensystem Routines — EISPACK Guide,* Springer-Verlag, Berlin (1976).

58. S. Wilson, Integral transformations, in: *Methods of computational chemistry, Vol. 1* (S. Wilson, ed), , Plenum Press, New York, (1987).

59. H.-J. Werner, Matrix-formulated direct MCSCF and multireference CI methods, *Adv. Chem. Phys.* **69**, 1-62 (1987).

60. W. Meyer, R. Ahlrichs, and C. E. Dykstra, The method of self-consistent electron pairs. A matrix oriented CI study, in: *Advanced Theories and Computational Approaches to the Electronic Structure of Molecules* (C. E. Dykstra, ed), 19-38, Reidel, Dordrecht, (1984).

61. I. Shavitt, The method of configuration interaction, in: *Modern Theoretical Chemistry, Vol. 3 Methods of electronic structure theory* (H. F. Schaefer, ed), 189-275, Plenum Press, New York, (1977).

62. P. D. Dacre, On the use of symmetry in SCF calculations, *Chem. Phys. Lett.* **7**, 47-48 (1970).

63. M. Elder, Use of molecular symmetry in SCF calculations, *Int. J. Quantum. Chem.* **7**, 75-83 (1973).

64. M. Dupuis and H. F. King, Molecular symmetry and closed-shell SCF calculations, *Int. J. Quantum. Chem.* **11**, 613-625 (1977).

65. M. Yoshimine, in IBM Report RA-18 (ed. W. Lester, 1971).

66. C. F. Bender, Integral transformations. A bottleneck in molecular quantum mechanical calculations, *J. Comput. Phys.* **9**, 547-554 (1972).

67. C. W. Bauschlicher, S. R. Langhoff, P. R. Taylor, N. C. Handy and P. J. Knowles, Benchmark full CI calculations on HF and NH_2, *J. Chem. Phys.* **85**, 1469-1474 (1986).

68. C. W. Bauschlicher, S. R. Langhoff, and P. R. Taylor, Adv. Chem. Phys., to be published.

69. E. R. Davidson, The iterative calculation of a few of the lowest eigenvalues and corresponding eigenvectors of large real symmetric matrices, *J. Comput. Phys.* **17**, 87-94 (1975).

70. B. Liu, The simultaneous expansion method for the iterative solution of several of the lowest eigenvalues and corresponding eigenvectors of large real-symmetric matrices, in: *Report on the NRCC Workshop on Numerical Algorithms in Chemistry: Algebraic Methods* (C. Moler and I. Shavitt, ed), 49-53, NRCC, Berkeley, (1978).

71. P. Siegbahn, A new direct CI method for large CI expansions in a small orbital space, *Chem. Phys. Lett.* **109**, 417-423 (1984).

72. P. J. Knowles and N. C. Handy, A new determinant-based full configuration-interaction method, *Chem. Phys. Lett.* **111**, 315-321 (1984).

73. C. W. Bauschlicher and P. R. Taylor, Benchmark full CI calculations on H_2O, F, and F^-, *J. Chem. Phys.* **85**, 2779-2783 (1986).

74. C. W. Bauschlicher and P. R. Taylor, Benchmark full CI calculations for several states of the same symmetry, *J. Chem. Phys.* **86**, 2844-2848 (1987).

75. P.-Å. Malmqvist, A. P. Rendell, and B. O. Roos, to be published.

76. M. A. Robb and U. Niazi, The unitary group approach in configuration interaction methods, *Comput. Phys. Rep* **1**, 127-236 (1984).

77. B. O. Roos and P. E. M. Siegbahn, The direct configuration interaction method from molecular integrals, in: *Modern Theoretical Chemistry, Vol. 3 Methods of electronic structure theory* (H. F. Schaefer, ed), 277-318, Plenum Press, New York, (1977).

78. P. E. M. Siegbahn, Generalizations of the direct CI method based on the graphical unitary group approach I. Single replacements from a complete CI root function of any spin, first-order wave functions, *J. Chem. Phys.* **70**, 5391-5397 (1979).

79. P. E. M. Siegbahn, Generalizations of the direct CI method based on the graphical unitary group approach II. Single and double replacements from any set of reference configurations, *J. Chem. Phys.* **72**, 1647-1656 (1980).

80. I. Shavitt, Matrix element evaluation in the unitary group approach to the electron correlation problem, *Int. J. Quantum Chem. Symp.* **12**, 5-32 (1979).

81. B. Liu and M. Yoshimine, The ALCHEMY configuration interaction method. I. The symbolic matrix method for determining elements of matrix operators, *J. Chem. Phys.* **74**, 612-616 (1981).

82. P. J. Knowles and H.-J. Werner, An efficient method for the evaluation of coupling coefficients in configuration interaction calculations, *Chem. Phys. Lett.* **145**, 514-522 (1988).

83. P. E. M. Siegbahn, unpublished work.

84. P. R. Taylor, A rapidly convergent CI expansion based on several reference configurations, using optimized correlating orbitals, *J. Chem. Phys.* **74**, 1256-1270 (1981).

85. R. Ahlrichs, A new MR-CI(SD) technique, in: *Proceedings of the 5th seminar on computational methods in quantum chemistry, Groningen, 1981* (P. Th. van Duijnen and W. C. Nieuwpoort, ed), 254-272, Max-Planck-Institut, Garching bei München, (1981).

86. V. R. Saunders and J. H. van Lenthe, The direct CI method: a detailed analysis, *Mol. Phys.* **48**, 923-954 (1983).

87. B. O. Roos, The CASSCF method and its application in electronic structure calculations, *Adv. Chem. Phys.* **69**, 399-445 (1987).

88. P. E. M. Siegbahn, J. Almlöf, A. Heiberg, and B. O. Roos, The complete active space SCF (CASSCF) method in a Newton-Raphson formulation with application to the HNO molecule, *J. Chem. Phys.* **74**, 2384-2396 (1981).

89. B. O. Roos, P. R. Taylor, and P. E. M. Siegbahn, A complete active space SCF method (CASSCF) using a density matrix formulated super-CI approach, *Chem. Phys.* **48**, 157-173 (1980).

90. B. O. Roos, The complete active space SCF method in a Fock-matrix-based super-CI formulation, *Int. J. Quantum Chem. Symp.* **14**, 175-189 (1980).

91. B. O. Roos, unpublished work.

92. J. Almlöf and H. P. Lüthi, Theoretical methods and results for electronic structure calculations on very large systems, in: *Supercomputer Research in Chemistry and Chemical Engineering. ACS Symposium Series 353* (K. F. Jensen and D. G. Truhlar, ed), 35-48, American Chemical Society, Washington, D. C., (1987).

93. T. U. Helgaker, J. Almlöf, H. J. Aa. Jensen, and P. Jørgensen, Molecular Hessians for large-scale MCSCF wave functions, *J. Chem. Phys.* **84**, 6266-

6279 (1986).

94. T. U. Helgaker, H. J. Aa. Jensen, P. Jørgensen, and P. R. Taylor, ABACUS, an MCSCF analytic energy derivatives program.

95. J. Almlöf, K. Faegri, and K. Korsell, Principles for a direct SCF approach to LCAO-MO ab initio calculations, *J. Comput. Chem.* **3**, 385-399 (1982).

96. M. Feyereisen and J. Almlöf, unpublished work.

97. M. Häser and R. Ahlrichs, private communication.

98. P. R. Taylor, Integral processing in beyond-Hartree-Fock calculations, *Int. J. Quantum Chem.* **31**, 521-534 (1987).

99. S. Saebø and J. Almlöf, *Chem. Phys. Lett.*, in press.

100. M. Head-Gordon, J. A. Pople, and M. Frisch, to be published.

101. B. Liu and A. D. McLean, *Ab initio* potential curve for Be_2 from the interacting correlated fragments method, *J. Chem. Phys.* **72**, 3418-3419 (1980).

102. M. R. A. Blomberg and P. E. M. Siegbahn, The ground-state potential curve for F_2, *Chem. Phys. Lett.* **81**, 4-13 (1981).

103. K. Jankowski, R. Becherer, P. Scharf, H. Schiffer and R. Ahlrichs, The impact of higher polarization basis functions on molecular *ab initio* results. I. The ground state of F_2, *J. Chem. Phys.* **82**, 1413-1419 (1985).

104. S. R. Langhoff, C. W. Bauschlicher, and P. R. Taylor, Accurate ab initio calculations for the ground states of N_2, O_2, and F_2, *Chem. Phys. Lett.* **135**, 543-548 (1987).

105. C. W. Bauschlicher, S. R. Langhoff, and P. R. Taylor, On the $^1A_1 - {}^3B_1$ separation in CH_2 and SiH_2, *J. Chem. Phys.* **87**, 387-391 (1987).

106. D. L. Cochrane and D. G. Truhlar, Strategies and performance norms for efficient utilization of vector pipeline computers as illustrated by the classical mechanical simulation of rotationally inelastic collisions, *Parallel Computing* **6**, 63-88 (1988).

107. D. W. Schwenke, Calculations of rate constants for the three-body recombination of H_2 in the presence of H_2, *J. Chem. Phys.* **89**, 2076-2091 (1988).

108. R. Bulirsch and J. Stoer, Numerical treatment of ordinary differential equations by extrapolation methods, *Numer. Math.* **8**, 1-13 (1966).

109. J. T. Muckerman, Applications of classical trajectory techniques to reactive scattering, *Theor. Chem.: Adv. Perspec.* **6A**, 1-77 (1981).

110. H. Goldstein, *Classical Mechanics,* Addison-Wesley, Reading (1980).

111. L. L. Poulsen, G. D. Billing, and J. I. Steinfeld, Temperature dependence of HF vibrational relaxation, *J. Chem. Phys.* **68**, 5121-5127 (1978).

112. W. A. Lester, Coupled-channel studies of rotational and vibrational energy transfer by collision, *Adv. Quantum Chem.* **9**, 199-214 (1975).

113. M. S. Child, *Molecular Collision Theory,* Academic Press, London (1984).

114. J. M. Blatt and L. C. Biedenharn, The angular distribution of scattering and reaction cross sections, *Rev. Mod. Phys.* **24**, 258-272 (1952).

115. D. G. Truhlar, C. A. Mead, and M. A. Brandt, Time-reversal invariance representations for scattering wavefunctions, symmetry of the scattering matrix, and differential cross sections, *Adv. Chem. Phys.* **33**, 295-344 (1975).

116. W. A. Lester, Calculation of cross sections for rotational excitation of diatomic molecules by heavy particle impact: solution of the close-coupling equations, *Meth. Comp. Phys.* **10**, 211-241 (1973).

117. Meth. Comp. Phys. **10**, (1971).

118. Comp. Phys. Commun. **6**, no. 6, (1973).

119. *Algorithms and Computer Codes for Atomic and Molecular Quantum Scattering Theory (L. Thomas,ed.),,* National Resource for Computation in Chemistry, Lawrence Berkeley Laboratory, Berkeley (1979).

120. M. Alexander, Hybrid quantum scattering algorithms for long-range potentials, *J. Chem. Phys.* **81**, 4510-4516 (1984).

121. M. H Alexander and D. E. Manolopoulos, A stable linear reference potential algorithm for solution of the quantum close-coupled equations in molecular scattering theory, *J. Chem. Phys.* **86**, 2044-2050 (1987).

122. D. W. Schwenke and D. G. Truhlar, A new potential energy surface for vibration-vibration coupling in HF-HF collisions. Formulation and quantal scattering calculations, *J. Chem. Phys.* **88**, 4800-4813 (1988).

123. D. W. Schwenke, A new method for the direct calculation of resonance parameters with application to the quasibound states of the H_2 $X\ {}^1\Sigma_g^+$ system, *Theoret. Chim. Acta* **74**, 381-402 (1988).

124. J. C. Light and R. B. Walker, An R-matrix approach to the solution of coupled equations for atom-molecule reactive scattering, *J. Chem. Phys.* **65**, 4272-4282 (1976).

125. E. B. Stechel, R. B. Walker, and J. C. Light, R-matrix solution of coupled equations for inelastic scattering, *J. Chem. Phys.* **69**, 3518-3531 (1978).

126. D. G. Truhlar, N. M. Harvey, K. Onda, and M. A Brandt, Applications of Close-Coupling Algorithms to Electron-Atom, Electron-Molecule, and Atom-Molecule Scattering, in: *Algorithms and Computer Codes for Atomic and Molecular Quantum Scattering Theory* (L. Thomas, ed), 220-289, National Resource for Computation in Chemistry, Lawrence Berkeley Laboratory, Berkeley, (1979).

127. G. H. Golub and C. F. Van Loan, *Matrix Computations,* Johns Hopkins

University Press, Baltimore (1983).

128. S. T. Elbert, Extracting more than a few eigenvectors from a dense real symmetric matrix: Optimal algorithms versus the architectural constraints of the FPS-X64, *Theoret. Chim. Acta* **71**, 169-186 (1987).

129. J. J. Dongarra, L. Kaufman, and S. Hammarling, *Squeezing the Most out of Eigenvalue Solvers on High-Performance Computers,* Argonne National Laboratory, Mathematics and Computer Science Division, Technical Memorandum No. 46 (1985).

130. D. W. Schwenke and D. G. Truhlar, Converged calculations of rotational excitation and V-V energy transfer in the collision of two molecules, in: *Supercomputer Simulations in Chemistry* (M. Dupuis, ed), 165-197, Springer-Verlag, Berlin, (1986).

131. J. J. Dongarra, C. B. Moler, J. R. Bunch, and G. W. Stewart, *LINPACK Users' Guide,* SIAM, Philadelphia (1979).

132. W. A. Lester, DeVogelaere's method, in: *Algorithms and Computer Codes for Atomic and Molecular Quantum Scattering Theory* (L. Thomas, ed), 105-115, National Resource for Computation in Chemistry, Lawrence Berkeley Laboratory, Berkeley, (1979).

133. M. E. Riley and A. Kuppermann, Vibrational energy transfer in collisions between diatomic molecules, *Chem. Phys. Lett.* **1**, 537-538 (1968).

134. D. W. Schwenke, K. Haug, D. G. Truhlar, R. H. Schweitzer, J. Z. H. Zhang, Y. Sun and D. J. Kouri, Storage management strategies in large-scale quantum dynamics calculations, *Theoret. Chim. Acta* **72**, 237-251 (1987).

135. J. M. Launay, Body-fixed formulation of rotational excitation: exact and centrifugal decoupling results for CO-He, *J. Phys. B* **9**, 1823-1838 (1976).

136. D. W. Schwenke and D. G. Truhlar, An optimized quadrature scheme for matrix elements over the eigenfunctions of general anharmonic potentials, *Comp. Phys. Commun.* **34**, 57-66 (1984).

137. N. A. Mullaney and D. G. Truhlar, The use of rotationally and orbitally adiabatic basis functions to calculate rotational excitation cross sections for atom-molecule collisions, *Chem. Phys.* **39**, 91-104 (1979).

138. J. M. Launay, Molecular collision processes I. Body-fixed theory of collisions between two systems with arbitrary angular momenta, *J. Phys. B* **10**, 3665-3672 (1977).

139. G. Gioumousis and C. G. Curtiss, Molecular collisions. II. Diatomic molecules, *J. Math. Phys.* **2**, 96-104 (1961).

140. D. W. Schwenke, D. G. Truhlar, and M. E. Coltrin, Comparison of close coupling and quasiclassical trajectory calculations for rotational energy transfer in the collision of two HF molecules on a realistic potential energy surface, *J. Chem. Phys.* **87**, 983-992 (1987).

141. J. V. Lill, G. A. Parker, and J. C. Light, Discrete variable representations and sudden models in quantum scattering theory, *Chem. Phys. Lett.* **89**, 483-489 (1982).

142. D. G. Truhlar and A. Kuppermann, Exact tunneling calculations, *J. Am. Chem. Soc.* **93**, 1840-1851 (1971).

143. J. Z. H. Zhang, D. J. Kouri, K. Haug, D. W. Schwenke, Y. Shima, and D. G. Truhlar, $\mathcal{L}^2$ amplitude density method for multichannel inelastic and rearrangement collisions, *J. Chem. Phys.* **88**, 2492-2512 (1988).

144. D. W. Schwenke, K. Haug, M. Zhao, D. G. Truhlar, Y. Sun, J. Z. H. Zhang, and D. J. Kouri, Quantum mechanical algebraic variational methods for inelastic and reactive molecular collisions, *J. Phys. Chem.* **92**, 3202-3216 (1988).

145. D. W. Schwenke, M. Mladenovic, M. Zhao, D. G. Truhlar, Y. Sun, and D. J. Kouri, Computational strategies and improvements in the linear algebratic variational approach to rearrangement scattering, to be published in the Proceedings of the NATO Advanced Research Workshop *Supercomputer Algorithms for Reactivity, Dynamics, and Kinetics of Small Molecules*, Colembella di Perugia, Italy, 1988.

146. D. G. Truhlar, J. Abdallah, and R. L. Smith, Algebraic variational methods in scattering theory, *Adv. Chem. Phys.* **25**, 211-293 (1974).

147. G. Staszewska and D. G. Truhlar, Convergence of $\mathcal{L}^2$ methods for scattering problems, *J. Chem. Phys.* **86**, 2793-2804 (1987).

148. J. Z. H. Zhang, S.-I. Chu, and W. H. Miller, Quantum scattering via the S-matrix version of the Kohn variational principle, *J. Chem. Phys.* **88**, 6233-6239 (1988).

149. W. H. Miller, Coupled equations and the minimum principle for collisions of an atom and a diatomic molecule, including rearrangements, *J. Chem. Phys.* **50**, 407-418 (1969).

150. D. W. Schwenke, D. G. Truhlar, and D. J. Kouri, Propagation method for the solution of the arrangement-channel coupling equations for reactive scattering in three dimensions, *J. Chem. Phys.* **86**, 2772-2786 (1987).

151. R. T. Pack and G. A. Parker, Quantum reactive scattering in three-dimensions using hyperspherical (APH) coordinates. Theory, *J. Chem. Phys.* **87**, 3888-3921 (1987).

152. G. Schatz, Quantum reactive scattering using hyperspherical coordinates: results for $H+H_2$ and Cl+HCl, *Chem. Phys. Lett.* **150**, 92-98 (1988).

153. L. Thomas, Solution of the coupled equations of inelastic atom-molecule scattering for a single initial state. II. Use of nondiagonal matrix Green functions, *J. Chem. Phys.* **76**, 4925-4931 (1982).

154. B. I. Schneider and L. A. Collins, Direct iteration-variation method for scattering problems, *Phys. Rev. A* **33**, 2970-2981 (1986).

155. C. Duneczky, R. E. Wyatt, D. Chatfield, K. Haug, D. W. Schwenke, D. G. Truhlar, Y. Sun, and D. J. Kouri, Iterative methods for solving the non-sparse equations of quantum mechanical reactive scattering, *Comp. Phys. Commun.*, in press

156. J. Abdallah and D. G. Truhlar, Use of contracted basis functions in algebraic variational scattering calculations, *J. Chem. Phys.* **61**, 30-36 (1974).

157. Y. Sun, R. C. Mowrey, and D. J. Kouri, Spherical wave close-coupling wave packet formalism for gas phase nonreactive atom-diatom collisions, *J. Chem. Phys.* **87**, 339-349 (1987).

158. S. Cray, presentation at CRI annual general meeting, 1988.

C-2

Table 1. SAXPY performance (in MFLOPS) on CRAY X–MP/48.

N[a]	CFT	CFT77	SCILIB
4	5.6	11.4	4.8
8	10.9	22.7	9.4
12	16.1	34.1	13.8
16	21.1	44.7	18.6
25	30.4	68.0	26.9
32	35.6	86.3	31.6
63	59.7	132.8	56.9
64	58.3	128.6	54.5
127	71.6	119.3	69.1
128	80.8	135.5	72.7
255	95.6	131.0	100.7
256	108.0	121.2	105.2
r_∞ ($n_{1/2}$)	110 (56)	159 (30)	118 (78)

[a] Vector length.

Table 2. SAXPY performance (in MFLOPS) on Cray computers .

N[a]	X–MP/14se[b]	X–MP/48[b]	Y–MP/832[b]	CRAY–2*[c]	CRAY–2[c]
4	11.1	11.4	15.2	3.7	3.3
8	22.1	22.7	30.3	7.0	6.5
12	33.2	34.1	45.5	10.1	9.5
16	44.2	44.7	60.6	13.0	12.1
25	69.0	68.0	94.5	19.1	18.8
32	84.8	86.3	120.9	23.2	22.7
63	112.9	132.8	211.5	40.3	38.1
64	103.3	128.6	213.5	42.6	37.6
127	104.4	119.3	213.1	41.1	53.4
128	103.9	135.5	214.1	52.2	50.8
255	119.6	131.0	242.1	58.0	67.8
256	104.0	121.2	242.5	80.5	62.2
300	120.5	158.6	178.7	70.8	66.2
511	121.7	143.3	206.8	63.0	45.1
512	120.3	140.6	206.5	67.2	63.1
r_∞ ($n_{1/2}$)	122 (22)	159 (30)	243 (32)	81 (63)	68 (55)

[a] Vector length.
[b] CFT77 gives best performance.
[c] SCILIB routine gives best performance.

Table 3. Matrix multiplication performance (in MFLOPS) on CRAY X–MP/48.

N[a]	DOT[b]	SAXPY[c]	4*unrolled[d]	MXM[e]
4	2.2	12.1	9.6	22.8
8	4.6	24.3	22.2	72.4
10	5.7	29.4	27.0	92.8
12	6.8	34.1	33.6	111.4
16	9.0	40.5	42.9	134.5
25	14.3	56.8	63.1	159.9
32	17.2	67.0	75.0	172.3
50	25.8	80.4	98.0	186.2
63	33.3	91.5	103.8	190.8
64	22.5	91.7	107.0	191.3
75	31.7	87.1	96.5	180.1
100	38.6	98.5	118.2	189.9
127	53.9	113.3	127.5	194.2
128	30.2	110.8	125.0	194.3
175	70.5	120.1	131.0	194.0
255	82.5	116.2	131.9	195.8
256	39.8	117.9	135.3	195.8
400	83.0	122.2	137.8	194.8
511	103.5	124.3	139.5	196.5
512	40.8	126.3	143.7	196.6
r_∞ ($n_{1/2}$)	104 (127)	126 (127)	144 (31)	197 (11)

[a] Vector length.
[b] Dot product inner loop; CFT77.
[c] SAXPY inner loop; CFT77.
[d] Unrolled to a depth of four (see text); CFT77.
[e] SCILIB routine.

Table 4. Matrix multiplication performance (in MFLOPS) on Cray computers.

N[a]	X–MP/14se	X–MP/48	Y–MP/832	CRAY–2*	CRAY–2
4	21.8	22.8	32.1	19.7	18.5
8	68.0	72.4	101.8	63.1	58.5
10	88.1	92.8	133.2	86.4	82.2
12	105.0	111.4	159.7	104.8	98.2
16	130.8	134.5	201.7	146.5	132.1
25	153.9	159.9	239.3	203.6	178.1
32	164.3	172.3	256.8	249.4	223.4
50	177.1	186.2	277.6	312.8	301.3
63	181.8	190.8	285.3	339.6	330.6
64	181.7	191.3	285.8	361.7	329.6
75	171.9	180.1	268.4	288.2	255.5
100	180.6	189.9	283.1	334.6	295.5
127	184.6	194.2	289.8	380.3	337.9
128	184.7	194.3	290.0	361.8	330.9
175	184.3	194.0	289.1	337.6	321.5
255	186.1	195.8	292.0	382.9	235.3
256	186.1	195.8	292.1	337.9	256.7
400	185.1	194.8	290.4	340.3	264.5
511	186.8	196.5	293.1	379.0	238.6
512	186.1	196.6	293.1	342.9	241.2
r_∞ ($n_{1/2}$)	187 (10)	197 (11)	293 (11)	383 (33)	338 (22)

[a] Vector length.

Table 5. Performance of specialized matrix multiplication techniques (in MFLOPS).

	CRAY X-MP/48		CRAY-2*			
N[a]	MXM[b]	8*unrolled[c]	MXM[b]	8*unrolled[c]	Strassen[d]	MXMPMA[e]
16	136	52	111	56	93	110
63	191	95	319	110	364	370
64	191	103	285	168	367	373
127	194	111	289	187	385	386
128	194	112	381	225	383	378
255	196	112	259	188	386	387
256	196	113	304	233	385	383
511	197	117	387	242	407	387
512	197	117	328	259	423	387

[a] Vector length.
[b] SCILIB routine.
[c] Unrolled to a depth of eight (see text); CFT2.
[d] Strassen algorithm, (Bailey, Ref. [27]).
[e] Calahan *et al.*, Ref. [23].

Table 6. Performance (in MFLOPs) for different operations[a].

Operation	X–MP/14se	X–MP/48	Y–MP/832	CRAY–2	CRAY–2*
Vector add	60 (24)	70 (24)	119 (28)	39 (63) L	48 (72) L
Dot product	90 (96)	95 (91)	148 (102)	80 (132) L	103 (96) L
SAXPY	122 (22)	159 (30)	254 (31)	68 (55) L	101 (128) L
Vector divide[b]	25 (16)	26 (15)	41 (16)	24 (23)	26 (26)
MXV[c]	187 (17) L	195 (18) L	312 (19) L	242 (27) L	294 (27) L
MXM[d]	187 (10) L	197 (11) L	308 (12) L	338 (22) L	383 (33) L
Cubic[e]	168 (17)	174 (15)	288 (19)	113 (15)	121 (17)
Sextic[e]	178 (14)	187 (13)	297 (16)	142 (17)	148 (17)
[2/1][f]	104 (13)	109 (12)	170 (13)	116 (16)	118 (17)
[3/2][f]	125 (11)	132 (11)	207 (12)	154 (16)	158 (17)
square root[b]	11 (15)	12 (16)	14 (15)	23 (21)	24 (19)
sine[b]	3 (9)	4 (10)	5 (11)	6 (13)	6 (16)
arctangent[b]	5 (11)	5 (11)	8 (13)	6 (16)	6 (15)
exponential[b]	6 (12)	6 (12)	9 (13)	9 (25)	9 (25)
$\log_e$[b]	4 (11)	4 (11)	6 (11)	6 (16)	6 (15)
MOVE[g]	76 (23)	83 (25)	143 (30)	51 (37)	72 (47)
GATHER[h]	26 (10)	46 (14)	80 (19)	17 (22)	20 (25)
SCATTER[h]	32 (11)	45 (13)	88 (20)	20 (16)	24 (16)
SPDOT[i]	32 (67) L	34 (64) L	52 (92) L	40 (230) L	48 (192) L
SPAXPY[i]	44 (58) L	47 (55) L	77 (66) L	31 (63) L	42 (96) L

[a] L denotes SCILIB routine, otherwise CFT77. $n_{1/2}$ values in parentheses.
[b] Performance in megaresults per second.
[c] Matrix-vector product.
[d] Matrix multiplication.
[e] Polynomial of given order with vector of arguments.
[f] Rational fraction of given order with vector of arguments.
[g] Vector move, performance in MW/s.
[h] Performance in MW/s.
[i] Sparse vector operations (see text).

Table 7. Macrotasked matrix multiplication performance (in MFLOPS)[a].

	1 CPU	2 CPUs	4 CPUs	8 CPUs
X–MP/48	200	399	796	
CRAY–2	420	756	1216	
Y–MP/832	293	585	1166	2320

[a] Obtained in stand-alone mode on X–MP/48 and CRAY–2. In stand-alone mode the Y–MP rates would be some 3–4% higher.

Table 8. Gaussian integrals and SCF timings for N_2 on CRAY X–MP/48[a] (in seconds).

	D_{2h}^b		D_{2h}		C_{2v}		C_s		C_1	
	N_{INT}^c	Time	N_{INT}	Time	N_{INT}	Time	N_{INT}	Time	N_{INT}	Time
Integrals	6230657	407.8	1837575	472.4	3426083	477.3	8416943	955.6	15741082	1676.3
					Ordering of symmetry integrals[d]					
Symmetry	N_{ORD}^e	Time	N_{ORD}	Time	N_{ORD}	Time	N_{ORD}	Time	N_{ORD}	Time
$\alpha\alpha\alpha\alpha$	509026	0.5	162710	0.2	1237527	0.6	6847625	12.6[f]	37271025	62.9[f]
$\alpha\beta\alpha\beta$	3411984	7.9[f]	1223503	1.2	3718896	6.1[f]	7840000	14.9[f]		
$\alpha\alpha\beta\beta$	940545	1.7	347383	0.7	995841	1.2	2037700	5.0[f]		
$\alpha\beta\gamma\delta$	3324672	10.2[f]	1321917	2.1	1249248	1.3				
Total	8186227	20.4	3055513	4.3	7201512	9.3	16725325	32.5	37271025	62.9
					SCF calculation					
	F[g]	Iter[h]	F	Iter	F	Iter	F	Iter	F	Iter
	0.4	0.6	0.2	0.4	0.4	0.7	0.6	1.2	1.0	2.2

[a] [5s 4p 3d 2f 1g] basis; spherical harmonic functions used except where indicated.
[b] Cartesian basis functions.
[c] Number of non-zero integrals computed.
[d] Ordering performed in memory unless otherwise specified.
[e] Number of ordered integrals generated.
[f] Ordering uses direct access storage (SSD).
[g] Closed-shell Fock matrix construction.
[h] Closed-shell SCF iteration time.

Table 9. Integral, SCF and transformation timings for N_2 (in seconds)[a].

		X–MP/48	CRAY–2*	Y–MP/832
		Integral calculation		
	N_{INT}^{b}			
Integrals	1837575	472.4	435.8	210.8
		Integral ordering		
Symmetry	N_{ORD}^{c}			
$\alpha\alpha\alpha\alpha$	162710	0.2	0.4	0.1
$\alpha\beta\alpha\beta$	1223503	1.2	2.7	0.6
$\alpha\alpha\beta\beta$	347383	0.7	1.5	0.4
$\alpha\beta\gamma\delta$	1321917	2.1	5.2	1.2
Total	3055513	4.3	9.9	2.3
		SCF calculation		
Fock matrix[d]		0.2	1.3	0.2
SCF iter[e]		0.4	1.4	0.3
		Integral transformation[f]		
Transformation		8.6	8.3	6.6

[a] [$5s$ $4p$ $3d$ $2f$ $1g$] spherical harmonic basis.
[b] Number of non-zero integrals computed in D_{2h} symmetry.
[c] Number of ordered integrals of each symmetry type. All ordering is done in memory.
[d] Closed-shell Fock matrix construction.
[e] Closed-shell SCF iteration.
[f] $1\sigma_g$ and $1\sigma_u$ MOs frozen in transformation.

Table 10. CI timings for N_2 on CRAY X-MP/48(in seconds)[a].

Calculation	Time
Formula tape for 16 internals[b]	< 0.1
Direct CI iteration (17 626 CSFs)	0.9
Formula tape for 2804 internals[c]	12.5
Direct CI iteration (729 950 CSFs)	79.7
Processing $(ai\|bj)$ integrals	38.3
Processing $(ai\|jk)$ integrals	18.1
Processing $(ab\|cd)$ integrals	10.2

[a] [5s 4p 3d 2f 1g] basis: $1\sigma_g$ and $1\sigma_u$ MOs frozen.
[b] Single reference CSF, 10 electrons correlated.
[c] CAS reference space (6 active electrons in 6 orbitals — 32 CSFs), 10 electrons correlated.

Table 11. Execution times and rates for classical trajectories.

	Fraction of total time				MFLOPS			
	X–MP/48[a]	Y–MP/832[b]	CRAY–2[c]	CRAY–2*[c]	X–MP/48	Y–MP/832	CRAY–2	CRAY–2*
			$F+H_2$					
Gradients of Potential	0.36	0.39	0.29	0.31	130	190	160	180
Build Q_n and Chain rule	0.27	0.24	0.20	0.20	120	210	160	190
sub total	0.63	0.64	0.49	0.51	126	200	160	184
Integrator	0.37	0.36	0.51	0.49	55	93	40	50
overall					100	160	99	118
			H_2+H_2					
Gradients of Potential	0.94	0.94	0.94	0.94	110	190	100	120
Build Q_n and Chain rule	0.03	0.03	0.02	0.03	140	210	140	160
sub total	0.97	0.97	0.96	0.97	111	190	101	121
Integrator	0.03	0.03	0.04	0.03	60	110	50	50
overall					110	190	98	119

[a] Using CFT compiler. The CFT77 compiler produces bad code for this problem.

[b] Using CFT77 compiler.

[c] Using CFT77 compiler. Due to the presence of some features of FORTRAN 77 which are only available under CFT77, we do not use CFT2.

Table 12. Characterization of the execution rates for matrix operations.

	Routine						
Machine	MXM[a]	RS[b]	LUSOLV[c]	LUSOLV[d]	MINV	SGEF A/SL[e]	SSPFA/SL[f]
X–MP/48	200[g]	170	190	210	73	130	73
	2	150	110	120	0	200	470
Y–MP/832	290	240	...	300	120	250	120
	1	100	...	110	7	150	470
CRAY–2	310	120	170	240	340	120	43
	4	25	160	290	80	140	330
CRAY–2*	370	140	210	200	360	140	50
	1	30	160	150	60	140	280

[a] matrix multiply routine from SCILIB.
[b] EISPACK [57] routine for real symmetric eigenvalues and eigenvectors from SCILIB.
[c] FORTRAN program for general linear equation solution [129,130] compiled using CFT2.
[d] FORTRAN program for general linear equation solution compiled using CFT77.
[e] LINPACK [131] program for general linear equation solution from SCILIB.
[f] LINPACK program for symmetric linear equation solution from SCILIB.
[g] Upper entry: estimate of asymptotic execution rate in MFLOPS, lower entry: fitted estimate (see text) of matrix order required to achieve half of the asymptotic rate.

/

Abstract

The influence of recent developments in supercomputing on computational chemistry is discussed with particular reference to Cray computers and their pipelined vector/limited parallel architectures. After reviewing Cray hardware and software ~~we examine~~ the performance of different elementary program structures WAS ~~was~~ examined, and ~~and outline~~ effective methods for improving program performance ~~were~~ are outlined. ~~We then discuss~~ the computational strategies appropriate for obtaining optimum performance in applications to quantum chemistry and dynamics ar ~~were~~ discussed. Finally, some discussion is given of new developments and future hardware and software improvements.